PREDICTING NONLINEAR TIME SERIES

THESIS

James C. Gainey Jr.
Captain, USAF

AFIT/GEO/ENG/93D-05

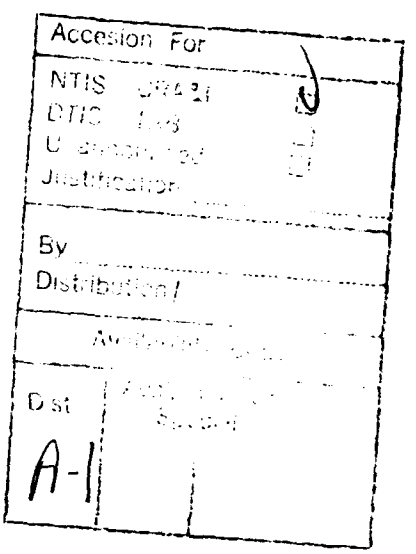
133 pgs

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GEO/ENG/93D-05

PREDICTING NONLINEAR TIME SERIES



THESIS

James C. Gainey Jr.
Captain, USAF

AFIT/GEO/ENG/93D-05

Approved for public release; distribution unlimited

AFIT/GEO/ENG/93D-05

PREDICTING NONLINEAR TIME SERIES

THESIS

PRESENTED TO

THE FACULTY OF THE GRADUATE SCHOOL OF ENGINEERING

OF THE AIR FORCE INSTITUTE OF TECHNOLOGY
AIR UNIVERSITY

IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

JAMES C. GAINEY, JR., B.S.E.E.

CAPTAIN

DECEMBER, 1993

ACKNOWLEDGMENTS

I came to AFIT with definite ideas about the topics I wanted to research. The capabilities of the human brain have fascinated me throughout my life. I give thanks to Dr. Steven K. Rogers for introducing me to neural network technology at a conference during my first assignment in the Air Force. I thank him even more now, as my thesis advisor, for the insight to help me scope this work and for the encouragement to work through the unforeseen obstacles. My desire to learn about the brain increases every time I discuss it with Dr. Rogers and Dr. Matthew Kabrisky. A special thanks goes to Dr. Kabrisky for his seemingly unlimited support and willingness to convey his knowledge. I owe a great deal of thanks to Capt Dennis Ruck, also, for his insight to neural networks and C programming. Thanks for teaching me the basics and helping me through the sticking points in implementing my code. I owe more thanks to my lovely and dedicated wife, Terri, than anyone else on Earth, though. She always picked me up when I was discouraged and pointed me back in the right direction. I have a lot to be thankful for, and I believe it was Jesus Christ that ultimately gave it all to me. I thank the Lord for giving His Son to me and for giving me the perseverance and knowledge to complete this learning experience.

James C. Gainey, Jr.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Predicting future values of a time series has many practical uses in real-time signal processing and understanding. This thesis implements an Adaptive Time Delay Neural Network (ATNN) capable of user-defined degeneration to the more common Time Delay Neural Network (TDNN). Time delays along axons or at the synapses, which vary in biological systems, motivate this research. The ATNN/TDNN test results and time series prediction capabilities are compared to those of the Real-Time Recurrent Learning (RTRL) algorithm. To show the advantages and disadvantages of using TDNN and ATNN for prediction versus the RTRL, the networks were applied to two problems: incommensurate sum of sine waves and financial time series. These data sets represent examples of nonlinear data with known and unknown mathematical functions, respectively. Although the RTRL predicted better than the ATNN for a known predictable function, this ATNN approach proved competitive in determining the direction of future values for this function and even outperforms the RTRL on the more difficult prediction task. The ATNN program, developed in C++ with an object-oriented framework, also takes much less computation time than the RTRL during training.

PREDICTING NONLINEAR TIME SERIES

I. INTRODUCTION

Most solutions to life's real world problems involve some sort of function prediction. Examining real world processes shows nature very rarely produces simple linear, easily mathematically modeled, functions. Many functions of practical interest even the basic sine wave are nonlinear. Humans possess an innate ability to assimilate information, make some sense of the information (probably in some nonlinear way of which little or nothing is known), and predict some useful outcome or required action. People constantly perform prediction involving trajectories, like catching a ball or avoiding a collision with other vehicles while driving, as background tasks. The human brain solves these real world problems easily, and in time to make the prediction useful, because they involve a relatively low order of dimensionality. A trajectory usually involves at most three dimensional time-space relationships. Nonlinear time series functions present more difficult real world problems, though. The dimensionality quickly becomes intractable even for the human brain. Many, seemingly extraneous, factors affect these functions forcing them towards unpredictability. Given a complex function separated into a superposition of more simple, predictable functions, the human brain's ability to predict completely diminishes when the complex time series function involves more than two sinusoidal functions with an irrational ratio of their frequencies (i.e., non periodic time series functions) [15]. People don't like to (actively) predict [7], especially these unfamiliar complex functions. For this reason almost all scientific disciplines pursue

mathematical or computer models to accurately predict the behavior of complex nonlinear processes inherent to their work.

Engineers trying to model real world processes, usually approximate these non-linear processes as a superposition of linearly separable functions. These nonlinear processes, or functions, are broken into as few linear parts as possible to still obtain acceptable results. By the time engineering approximations are made, too much information is sometimes thrown away and some engineers and scientists feel they must keep all that's left. Modeling of this type usually results in poor representations of the original functions because engineers model the smooth (nonlinear) transitions that nature makes by too few of these linear pieces. Meteorologists trying to predict the weather, economists trying to predict market behavior, and physicists attempting to track turbulent flow of fluids, all using ever faster supercomputers, frequently produce unsatisfactory results [5]. This strategy lasted through the years, probably, because it was the best available.

However, for today, even large numbers of supercomputers working in parallel can not accurately reproduce the processing capabilities of an incredible 3 pound piece of meat called the human brain. This biologically motivated processing may never be achievable by man-made machines, but the ideas behind Artificial Neural Networks (ANN) offer some hope. They possess, as might be imagined the brain does, the ability to map relationships in a nonlinear way. It is time for conventional engineers to rethink their "old" strategies. Today's technologies obtain an abundance of information quickly and easily, and current machines store and sort this historical information well. It is the information processing techniques that seem to be lacking. Maybe engineers should take a lesson from Mother Nature, keeping only the best parts, or features, of the information like in Nature's rule: **Survival of the Fittest**. Working smarter almost always produces better results

than working harder. These fittest features would store only important and often used information for later processing much as the human brain uses or loses memory.

Human brain development is an interesting phenomenon. Jerison points out,

> It's estimated that it took around one million years to make the human brain what it is today, given this is about the time when a rapid increase in brain size compared to body weight began in the hominid lineage. There is no evidence of a change in brain size versus body weight for any other mammals within the past five million years. [6]

The modeling of brain functions does not have to start at the beginning, though. Today's electronic, chemical, and imaging technology allows us to look deeper inside the human skull and into the brain. The human brain, most people think, is the most powerful information processor in creation. Therefore, try to more precisely model the best. In other words, if one models and strives for excellence, he is destined to achieve excellence [13], or better still

> *"...if you refuse to accept anything but the best, you very often get it."*
>
> *- W. Somerset Maugham*

Consider moving up to a better model of the real world; stop accepting methods of the past. Today's conventional sensors provide enough information for a human to predict an event outcome given relatively low dimensionality, but automatic real-time prediction, using today's computer architectures, is often too slow and computationally expensive. Thus, non real-time prediction, in most applications, equates to present state estimation, not to prediction. Webster [11] defines

> **to predict** - is to declare in advance; esp.: foretell on the basis of observation, experience, or scientific reason.

So, this thesis defines nonlinear function prediction as the declaration of a future value of the given function, based on that function's history.

Work performed by Capt Randall L. Lindsey [10] shows properly trained Recurrent Neural Networks are successful in applications involving time dependencies, including function prediction. A recently completed competition, at the Santa Fe Institute, also found Recurrent Neural Networks among the best, but a Time Delay Neural Network (TDNN) algorithm won the time series prediction competition [21]. Another recent paper discusses an Adaptive Time-delay Neural Network (ATNN) algorithm for prediction [9]. This seems a logical improvement over the TDNNs.

1.1 Problem

This thesis focuses on solving the nonlinear function prediction problem by developing an ATNN and TDNN program in C++. For various types of input data, the results show the comparison of these networks with the subgrouped RTRL in terms of testing and training accuracy versus training time.

1.2 Background

Neural network publications span multiple disciplines: neurobiology, physics, psychology, medical science, mathematics, computer science, and engineering. Since its revival in the mid 1980's, neural network technology has been changing too rapidly to include a complete summary of it here. Searching current databases for a more narrow review including the just a few important features for nonlinear function prediction yields a manageable review. This Literature Review, as presented in Chapter II, highlights the following features: Real-Time Recurrent Learning (RTRL) algorithms, subgrouped RTRL, TDNNs, and ATNNs.

ANN technology continues to improve by leaps and bounds. Meanwhile researchers and engineers apply the state-of-the-art algorithms to solve time-series prediction problems. The ANNs, above, when properly trained, solve many time series tasks and are useful applications.

1.3 Assumptions

This thesis assumes the input vectors are actual past time samples of the process under study. The only input feature available to the networks will be this historical time sequence data of the process itself. Therefore, feature extraction from an input pattern is not addressed here, but successful selection of the ANN parameters during the training and testing periods is essential for each algorithm. The processes are assumed to be predictable; that is, they are not purely random.

1.4 Scope

This thesis will present an Adaptive Time-delay Neural Network for time series prediction of complex functions and compare the results to the TDNN and the Real Time Recurrent Learning (RTRL) algorithm.

1.5 Approach

The proposed plan is to create a nonlinear function prediction capability in five steps. First, create the Adaptive Time-delay Neural Network program. This ATNN algorithm, coded in an object-oriented C++ programming framework, configures to the more conventional fixed Time-delay Neural Network (TDNN) or Error-Backpropagation (BP) algorithms with the appropriate user-defined network parameters. Second, implement a RTRL algorithm to solve a few specific time series function problems.

Third, modify the ATNN into a TDNN and train then test for solving time series prediction with fixed time delays. Fourth, train and test the full ATNN algorithm with adaptive time-delays as well as weights for solving the prediction problem. Finally, compare all the algorithms in terms of training and testing accuracy versus number of training cycles, or epochs.

This chapter provides a brief perspective, the goal of this thesis, and then outlines the approach to studying this prediction problem. The next chapter will review some of the background material essential to understanding the current ANNs at the forefront of complex function prediction. Chapter III develops the algorithm for the ATNN (and TDNN) while Chapter IV presents results of applying time series data to the RTRL network as well as the ATNN and TDNN. Chapter V presents conclusions and recommends direction of further study with these networks.

II. LITERATURE REVIEW

2.1 Introduction

This literature review summarizes the current state of artificial neural networks (ANN) for solving time dependent processes. Biological neural networks quickly and easily process temporal information; artificial neural networks should do the same. Modeled from biological research, artificial neural networks provide a heuristic approach to solving problems that could prove quite successful in areas of speech processing and image recognition [8]. Of the many known varieties of ANNs, literature suggests only a few of the major classes are adequate for difficult temporal processing and prediction tasks. One class, the time delay neural network (TDNN), incorporates embedded time delays on the inputs. Another, known as the recurrent neural network, uses time delayed network outputs that feedback as inputs to encode and learn temporal sequences. Time dependent processes govern much of the real world. Thus, properly trained artificial neural networks, both TDNN and recurrent, could prove very successful in applications involving time dependencies.

Publications in the field of neural networks span all the disciplines of science: neurobiology, physics, psychology, medical science, mathematics, computer science, and engineering. As such, a thorough summary of neural network technology would contain numerous volumes. However, a sampling of current literature, centered on the topics of time series prediction, TDNNs, and recurrent backpropagation neural networks, yields a more focused review.

The scope of this review focuses on current literature detailing studies of nonlinear time series function prediction. In particular, an even more narrow focus on uses of TDNNs and recurrent backpropagation neural network technology reveals a tremendous effort exists to solve problems of this nature. This review contains a short background on

basic neural network theory to aid the reader's comprehension. In addition, this review highlights the real-time recurrent learning (RTRL) algorithm, subgrouped RTRL, TDNN algorithm, and finally, an Adaptive Time Delay Neural Network (ATNN). These algorithms summarize improvements in neural network technology as applied to function prediction.

2.2 Background

Biological concepts motivate the application of artificial neural networks to electronic machines. Artificial neural networks attempt to copy or mimic the response of a true biological neuron, the most basic processing element of the brain [14].

During the late 1950's, Rosenblatt invented a new class of machines offering, as many researchers thought, a natural and powerful model of machine learning [16]. This basic model, called the perceptron, consists of an array of input sensory nodes randomly connected to a second array of associative nodes. The connections, called weights, randomly range from -1 to 1. Each secondary node produces an output when activated by enough of the sensory nodes connected to it. Sensory nodes capture outside information for the machine, and associative nodes input the information to the machine.

The output, or response, of the perceptron equals a proportional weighted sum of the associative nodes' responses. In other words, if x_i denotes the response of the ith associative node and w_i denotes the corresponding connection weight, then the activation S of the next node with n associative input nodes is

$$S_j = \sum_{i=1}^{n} w_{ji} x_i \qquad (2.1)$$

and the output, or response R, of this next node is given as

$$R = f\left(\sum_{i=1}^{n} w_{ji} x_i\right)$$

Thus for a positive R, the stimulus belongs to class 1, and for a negative R, the stimulus belongs to class 2. In its most basic form, the perceptron simply implements a linear decision function $f(x)$. The perceptron learns by changing the connection weights to minimize the total response error. The difference between the desired output and the actual computed response of the node defines the nodal error. In equation form,

$$e_n = d_n - R_n$$

where e_n denotes error of node n, and d_n denotes the desired value of node n. Therefore, the total response error equals the summation of the squared nodal errors over the entire length of the data set (epoch).

In most applications, a differentiable function, usually the logistic squashing function (sigmoid), operates on the output of the network. When the sigmoid response, given by

$$f(x) = \frac{1}{1+e^{-\alpha x}}, \tag{2.2}$$

operates on the input, the response becomes the weighted sum of the inputs, including a bias term θ. Thus the resulting output becomes

$$R_j = f_j\left(\sum_{i=1}^{n} w_{ji} x_i + \theta_j\right), \tag{2.3}$$

and Figure 1 shows a typical network node while Figure 2 details the output of the sigmoid.

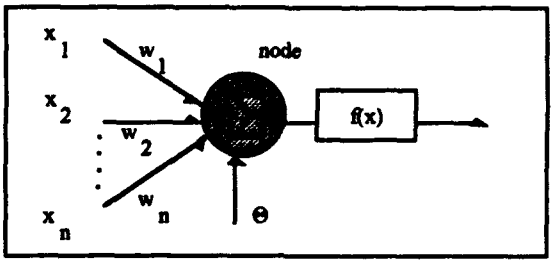


Figure 1: Rosenblatt's perceptron model.

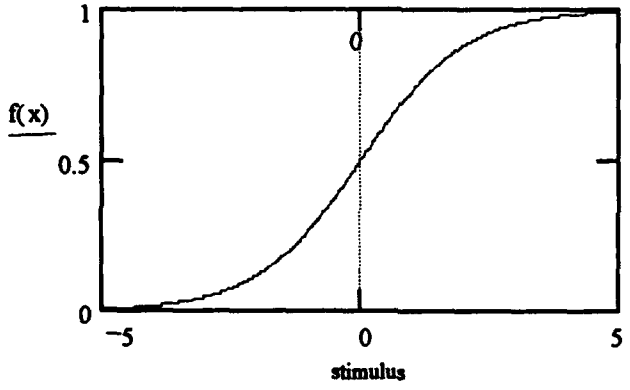


Figure 2: A typical sigmoid function with α =1.

Many other architectures propose extensions to this basic concept introduced by Rosenblatt. Multilayer perceptrons, feedforward neural networks, error-backpropagation networks, and recurrent backpropagation networks name just a few. Error-backpropagation, discovered by Werbos in 1974 [22] and independently by Parker in 1982 [12], refers to the method of updating the interconnection weights; that is by propagating backward from the output to the input and changing each connection weight minimizes the total error. Feedforward neural nets refer to connection schemes where the direction of information flow is strictly from input to the next layer passing forward through successive

layers to the output. Error passes backward to adjust the weights during training but does not add information to any given node. Think of error-backpropagation as weight modification rather than unit activation. In other words, no "information" flows (as input) from higher-level to lower-level nodes. Many texts contain the derivation of the error backpropagation algorithm[14]. A purely feedforward network would only react to external input. Feedforward artificial neural networks typically solve recognition problems by separating spatial regions. However, they can also "learn" relationships governing the generation of a time series, then use that "knowledge" to predict future values of the time series [17]. Building on that idea, recent time delay neural networks spatially present a fixed time "window" of the sequence to the network inputs. Thus, the TDNN architectures utilize a fixed number of the actual time sequence values as inputs instead of the recurrent network architecture where time delayed network outputs feedback to inputs.

Feedback, sometimes called internal input, endows a network with a couple of significant features: (a) incorporates multiple time scales into the processing nodes, (b) processes temporal sequences of inputs [3]. A recurrently connected neural network contains feedback loops from previous states (timed inputs) as well as the backpropagation. The next sequentially timed input uses these feedback outputs to create a predictive output at time $t + 1$ based on the current input and the previous output. Information about past inputs is manifest through the learning modified results in this previous output. As with the input vector, the feedback connections each have their own adaptable weights. These recurrent weights change, through backpropagation, to minimize the total error over the epoch length. Figure 3 shows a general layout of a recurrent neural network. Notice that the current input vector at time t includes a bias input (always equal to one), the external inputs, and the previous network's output. A

brief look at some of the work accomplished by applying various network designs to time series data follows.

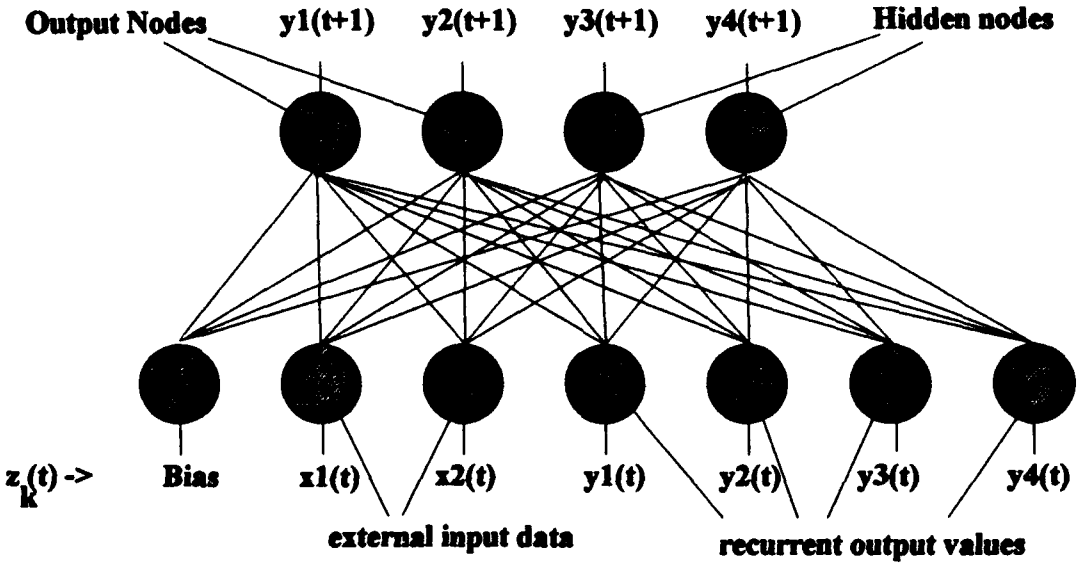


Figure 3: Basic RTRL architecture, with two outputs, two hidden nodes, and two inputs [10].

2.3 Real-Time Recurrent Learning (RTRL)

Williams and Zipser describe a real-time learning algorithm for training completely recurrent, continually updated networks to learn temporal tasks [23]. This technique uses uniform starting configurations that contain no previously known information about the temporal nature of the task. It presents a gradient-following learning algorithm that tracks the total network error along a trajectory minimizing this total error. Its two prime advantages include not requiring a precisely defined training interval and operating while the system is running. The algorithm's main disadvantage consists of requiring nonlocal communication during training making it computationally intensive. Yet, this algorithm, called the real-time recurrent learning (RTRL) algorithm, allows recurrently connected

networks to learn complex tasks that require the retention of information over definite or indefinite time periods.

2.4 *Subgrouped RTRL*

Whereas the previously discussed RTRL algorithm shows great power and generality, its disadvantage (CPU intensive) presents a potential problem. Zipser addresses this problem by proposing an improved technique which reduces the amount of computation time required by RTRL without changes in the network connectivity called network subgrouping [24]. Subnets, which divide the original network for the purpose of error backpropagation, leave the network undivided for forward propagation of activations. This means that, during training, subgroups form only when error propagates backward through the network's connection weights. During normal feedforward propagation, the network remains fully connected. This subgrouped RTRL algorithm is 10 times faster when compared to the previous RTRL method performing a specific learning task [24]. It suffers, however, in that each subgroup now has less memory than the original RTRL. Zipser suggests compensating for this by using more hidden nodes.

2.5 *Time Delay Neural Networks (TDNN)*

TDNNs have recently been applied for use in phoneme classification [18]. Figure 4 shows a typical TDNN architecture used for this classification problem. Waibel used this network, with some success, for the identification of phonemes in Japanese. However, work done in conjunction with a recent competition held at the Santa Fe Institute proves the usefulness of TDNNs for solving complex time series prediction tasks [21]. A Finite Impulse Response (FIR) neural network [19], equivalent to TDNN, won the competition with recurrent networks finishing close behind [21].

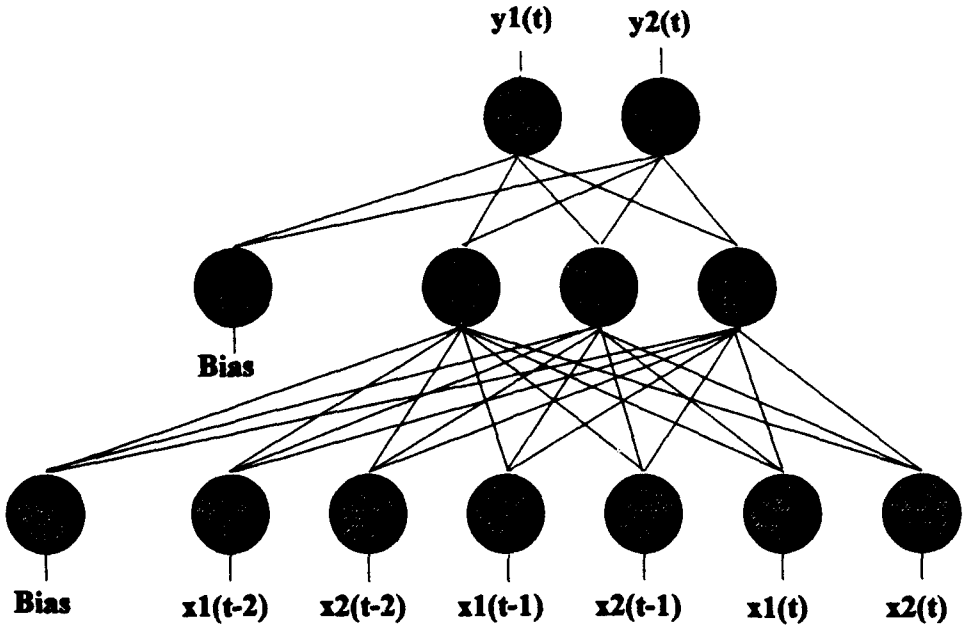


Figure 4: Typical TDNN network where input data is shifted along inputs to the net.

This approach models the biological synapse as a FIR linear filter. The article, also, derives a temporal generalization of the familiar error backpropagation algorithm. This feedforward TDNN method replaces the scalars of Equation 2.3 with vectors representing the weighted sum of delayed samples of the inputs. The response becomes

$$R_j(k) = f_j\left(\sum_{i=1}^{N_{i-1}} \vec{W}_{ij}\,\vec{X}_i(k)\right) \qquad (2.4)$$

where W_{ij} specifies the weight associated with the output of node i to the input of node j in the next layer and k is the discrete time index. The temporal backpropagation algorithm is similar to the Adaptive Time Delay Neural Network developed in Chapter III. This algorithm allows for more computational efficiency since the number of operations grows linearly with the number of layers in the network.

2.6 *Adaptive Time Delay Neural Networks (ATNN)*

Another article presents the logical next step to follow the TDNN [9]. It describes an improved learning algorithm based on gradient-descent for updating the time delays as well as the synaptic weights. The delays in a biological system occur along axons, due to factors such as axon length and insulation (myelin sheath), and at the synapses due to the biochemical processes. This adaptation method provides more flexibility so the network can attain the optimal time delays associated with the weight to achieve more accuracy than with fixed delays determined prior to training or by trial-and-error. As a result, an ATNN network proves slightly better than TDNN because of a better match between choice of time delay values and the temporal location of the important information in the input patterns. This thesis will focus on the development of C++ code to implement this ATNN algorithm and then compare the performance of it with TDNN, by fixing the weights of the ATNN, and subgrouped RTRL.

2.7 *Summary*

As technology improves, engineers and other researchers discover new and innovative algorithms for solving time-dependent problems. All of these algorithms discussed here possess the ability, if properly trained, to tackle and solve many difficult temporal tasks. More great strides in advancing neural network technology require still further research. Most of these algorithms exist in digital software form. Some routines cannot be implemented in contemporary hardware. Therefore, further research will determine whether or not particular networks become physical hardware elements, thus greatly increasing their speed and utility.

A theme that carries throughout this thesis concerns time and the necessity for network models to reflect the fundamental and essential temporal nature of actual nervous systems. Short-term memory allows present access to the recent past, and longer-term

memory relates to the more remote past. To respond to temporal processes, the nervous system may require a temporal representation. If so, ANNs should also be capable of representing processes extended in time. Captain Randall Lindsey investigated RTRL algorithms for predicting time series functions [10]. Captain Jeffrey Dean modified Lindsey's RTRL code for the subgrouped RTRL algorithm, which this thesis uses due to its increased performance. The next chapter develops an ATNN algorithm (and TDNN algorithm by user definitions) and code for comparison to the prediction abilities of the recurrent networks such as subgrouped RTRL.

III. METHODOLOGY

3.1 Introduction

The recent works covered in the Literature Review pertaining to time series function prediction, provides a broad overview of the types of artificial neural networks most commonly used in today's prediction research. This thesis effort encodes the Adaptive Time Delay Neural Network (ATNN). From this code, the user defines either an ATNN, Time Delay Neural Network (TDNN) or a Backpropagation network (BP) as desired. The resulting network gets tested in both the ATNN and TDNN modes for accurate time series function prediction. As stated earlier, a previous AFIT thesis by Lindsey encodes the Real-Time Recurrent Learning (RTRL) algorithm and yet another AFIT thesis, by Capt Jeffrey Dean currently near completion, modifies Lindsey's code extending it to the subgrouped RTRL case. Since the subgrouped RTRL algorithm is basically the same as RTRL but is less computationally intensive, this thesis compares subgrouped RTRL with the ATNN and TDNN prediction schemes developed here.

This methodology chapter develops the ATNN algorithm for performing time series function prediction. The explanation details the basic theory and interpretation of the ATNN algorithm used in this thesis. In addition, this chapter discusses how to use the code for ATNN learning versus TDNN or Error-Backpropagation Network learning (the later two are special cases of the more general ATNN algorithm). Finally, this chapter discusses the training and testing procedures and applies them to two specific problems.

3.2 ATNN Algorithm Development

This Adaptive Time Delay Neural Network (ATNN) generalizes the common error-backpropagation, gradient descent method to allow for adapting of time delays as well as the weights during training. Time delays allow each node in the artificial neural network

to more closely model what is understood of true biological processes. Studies show time delays exist in the biological nervous system due to impulse transmission, caused by the varied lengths and insulation of axons, and cell membrane excitation, caused by the temporal properties at synapses [4]. Time delay network nodes take into account not only the information from previous layers (as in standard feedforword networks), but they also remember some of the past information in the delays associated with each interconnection. This ATNN algorithm interpretation provides a variable length buffer (like a memory) for each input feature in the data set instead of the spatio-temporal separation method described for TDNNs in Section 2.5 (Figure 4).

3.2.1 NETWORK ARCHITECTURE

The interconnection scheme for ATNN nodes includes n connections between previous node i and the next node j. Each connection contains its own time delay and associated weight. Attempting to simplify the variable indexing, the order of indices remains constant throughout this thesis. The first index corresponds to the next node which the connection goes to, the second index corresponds to the previous node from which the connection comes, and the third index gives the particular time delay connection between the two nodes. If only one index appears on a variable, j relates to the output of the next node and i relates to the input from the previous node. In their article, Lin and others [9] call this interconnection scheme between nodes a delay block as shown in Figure 5 where τ_{jin} and w_{jin} are the independent time delays and weights for each nth interconnection of the block. Summation of the activations into each node occurs the same as in the basic Rosenblatt perceptron discussed earlier. However, now the activations result both from the input of previous nodes and the n time delay interconnections for each of these previous nodes.

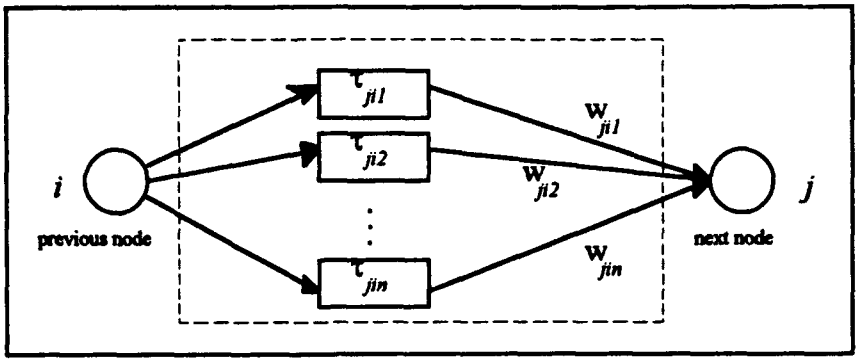


Figure 5: Delay block for ATNN [9]

Thus, node *j* receives the total activation given in the following equation

$$S_j(t_n) = \sum_{i=0}^{N} \sum_{k=1}^{K} w_{jik} \cdot a_i(t_n - \tau_{jik}) \qquad (3.1)$$

where N is the number of input features and K is a maximum number of time delay interconnections between nodes. In the equation, $a_i(t_n - \tau_{jik})$ is the activation from the output of a previous node i (or an external input sample) delayed by τ_{jik}. For simplicity, each node in the network then processes the weighted sum through the same sigmoid function as described by Equation 2.2.

A user definition file completely specifies the desired ATNN architecture. This file consists of basic variables used in most artificial neural network configurations (i.e., numbers of input (n), hidden (q), and output (p) nodes, weights update learning rate (η_1), maximum number of iterations through the training data (epochs), and range of random initialization values for matrices). The file also includes the ATNN specific variables (i.e., time delay update learning rate (η_2), momentum, maximum number of time delays (K), epoch learning flag - to update after each epoch instead of after each input pattern, tolerance - user defined acceptable difference between desired and network output, time interval between data points, and TDNN learning flag - for fixing time delays).

The encoded network structure allows for only one layer of hidden nodes, thus, two layers of weights and two layers of time delays. This ATNN architecture uses the delay block idea to connect between any two nodes and the nodes fully connect from one layer to every node in the next layer. This might not be biologically correct but the learned weight values in the ANN give relative strength to these interconnections. Thus, the unused interconnections obtain a very low weighting. Each connection can also have its own independent time delay. Figure 6 shows a two layer ATNN example (i.e., two layers of delay blocks) for clarity.

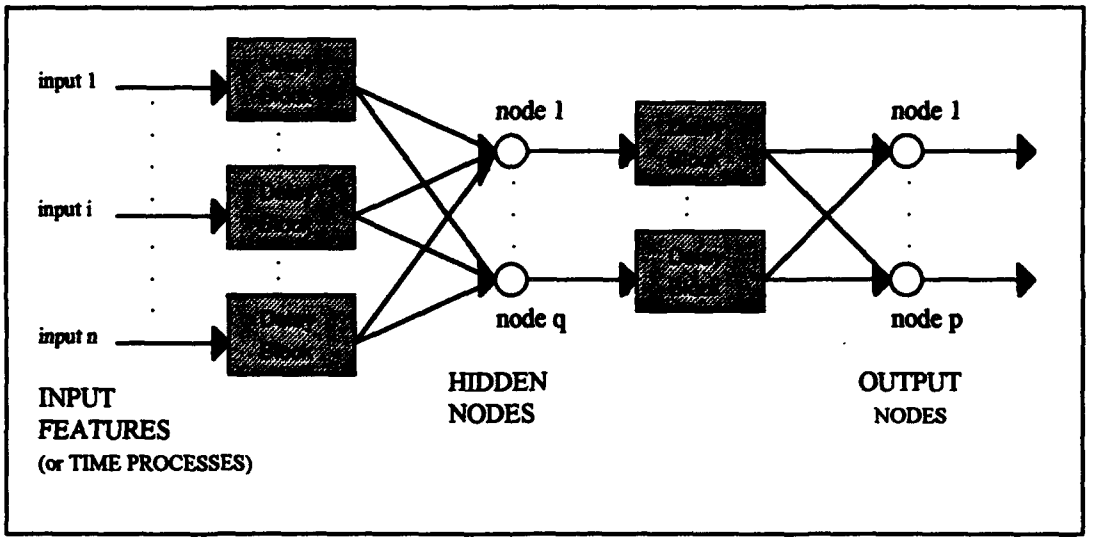


Figure 6: Two Layered ATNN [9]

3.2.2 Learning Algorithm

The derivation of the ATNN algorithm presented here follows that of the article by Lin [9]. This thesis shows only the most important equations for understanding the algorithm. Many texts contain the complete derivation of Error-Backpropagation weight update rule [14], and the article [9] derives the time delay update rule as found in Appendix A for completeness.

First define an instantaneous error measure (MSE) by

$$E(t_n) = \tfrac{1}{2}\sum_{j\in P}(d_j(t_n) - a_j(t_n))^2 \tag{3.2}$$

where there are P output nodes with computed values, $a_j(t_n)$, and $d_j(t_n)$ denotes the desired output of node j at time t_n. In this thesis, the weights and time delays update after each input pattern according to the respective error gradients

$$\Delta w_{jik} \equiv -\eta_1 \frac{\partial E(t_n)}{\partial w_{jik}} \tag{3.3}$$

$$\Delta \tau_{jik} \equiv -\eta_2 \frac{\partial E(t_n)}{\partial \tau_{jik}}. \tag{3.4}$$

where η_1 and η_2 are the learning rates.

Let

$$\delta_j(t_n) = \begin{cases} (d_j(t_n) - a_j(t_n))f'(S_j(t_n)) & \text{if } j \text{ is an output node} \\ (\sum_{p\in P}\sum_{q=1}^{K}\delta_p(t_n)w_{pjq}(t_n))f'(S_j(t_n)) & \text{if } j \text{ is a hidden node} \end{cases} \tag{3.5}$$

and

$$\rho_j(t_n) = \begin{cases} -(d_j(t_n) - a_j(t_n))f'(S_j(t_n)) & \text{if } j \text{ is an output node} \\ -[\sum_{p\in P}\sum_{q=1}^{K}\rho_p(t_n)w_{pjq}(t_n)]f'(S_j(t_n)) & \text{if } j \text{ is a hidden node} \end{cases} \tag{3.6}$$

where P now is the number of nodes in the next layer all terms are defined as before and K is the maximum number of time delays. Then the learning rules can be summarized as follows:

$$\Delta w_{jik} = \eta_1 \delta_j(t_n) a_i(t_n - \tau_{jik}) \tag{3.7}$$

$$\Delta \tau_{jik} = \eta_2 \rho_j(t_n) w_{jik} a_i'(t_n - \tau_{jik}). \tag{3.8}$$

Because the node activation varies with time, the error gradient with respect to time requires calculation of this derivative. According to the Mean Value theorem of differential calculus, the derivative of the node activation in Equation 3.8 approximates to

$$a_i'(t_n - \tau_{jik}) \approx \begin{cases} \dfrac{a(t_n) - a(t_{n-1})}{r} & \text{if } \tau_{jik} = 0 \\ \dfrac{a(t_{k+1}) - a(t_{k-1})}{2r} & \text{if } t_n - \tau_{jik} = t_k, \tau_{jik} \neq 0 \end{cases} \tag{3.9}$$

where r is the time step between data points. The time delays are, in general, zero or positive real numbers. However, here the update is based on time delays rounded to an integer number of time steps.

An example of how to interpret these learning rules will be helpful for the unfamiliar reader. For instance substituting Equation 3.6 into Equation 3.8 and dropping subscripts, the time delay update rule for a hidden layer simplifies to

$$\Delta\tau_{jik} = \eta_2 \sum_P \sum_K [(d-a)a(1-a)W2]h(1-h)W1(a').$$

The above reads as follows:

> the (learning rate) times the sum over all (output nodes) and (time delays) of [(desired - net output)*(net output)*(1-net output)*(weights in layer 2)] times (hidden activation) times (1- hidden activation) times (weights of layer 1) times (derivative of input features).

The other instances of learning rules follow a similar interpretation. Given this overview of the ATNN algorithm itself, the next section explains the code developed for this thesis effort.

3.3 *ATNN Code Development*

This ATNN code uses an object-oriented programming (OOP) structure in C++. C++ and OOP are becoming the standard for software development. The C++ flexibility and reusable "objects" provide an environment to more easily develop and implement various artificial neural networks. Even the C programming language does not yet offer a well-developed tool kit for implementing artificial neural network algorithms. Adam Blum developed an object-oriented framework, on which this ATNN code is based, for building neural networks in C++ [1]. C++ offers all the advantages of C but also supports object-oriented programming readily. Programming objects and using them to build a framework, much like a carpenter uses tools to build a frame house, allows for more flexible and innovative designs. Most of the applications and developments in Blum's book deal with spatial problem solving. This thesis effort, on the other hand, designs new methods and classes for representing time series data and solving time dependent process problems. The new objects allow incorporation of 3-dimensional matrices, buffered inputs to nodes, propagation and backpropagation through time, activation derivatives, and time delay optimization learning. A new class, ATNN, encompasses all these methods that operate together to form the ATNN algorithm. The ATNN class is a specialization of the net class developed by Blum. Figure 7 shows the class hierarchy for an ATNN.

Making the ATNN code executable requires several definition and source code files for the various classes used to build an ATNN. The definition files include: atnn.h, net.h, and vecmat.h. The source code files include: testatnn.cc, atnn.cc, net.cc, and vecmat.cc. Complete listings for the required files to compile and use the ATNN program are included as Appendix C. The ATNN code was developed using Turbo C++ 3.0 on an IBM/compatible 486 system. Although it successfully compiles using Turbo C++ 3.0, training is quite slow on this single processor system. For that reason, the ATNNs

described in this thesis were trained and tested using the Silicon Graphics systems (IRIS 4D and ONYX).

The details of using the interactive ATNN program are contained in Appendix B. The program is interactive in that it allows the user to set a maximum number of epochs or target minus network output tolerance for determining program completion, but the training process may be suspended (by pressing the ESC key on the Silicon Graphics or any key on a PC) and saved at any point. Subsequent training automatically resumes from the stored network information (located in the associated .WTS file). This allows for testing at different points during the training process which helps in optimizing training.

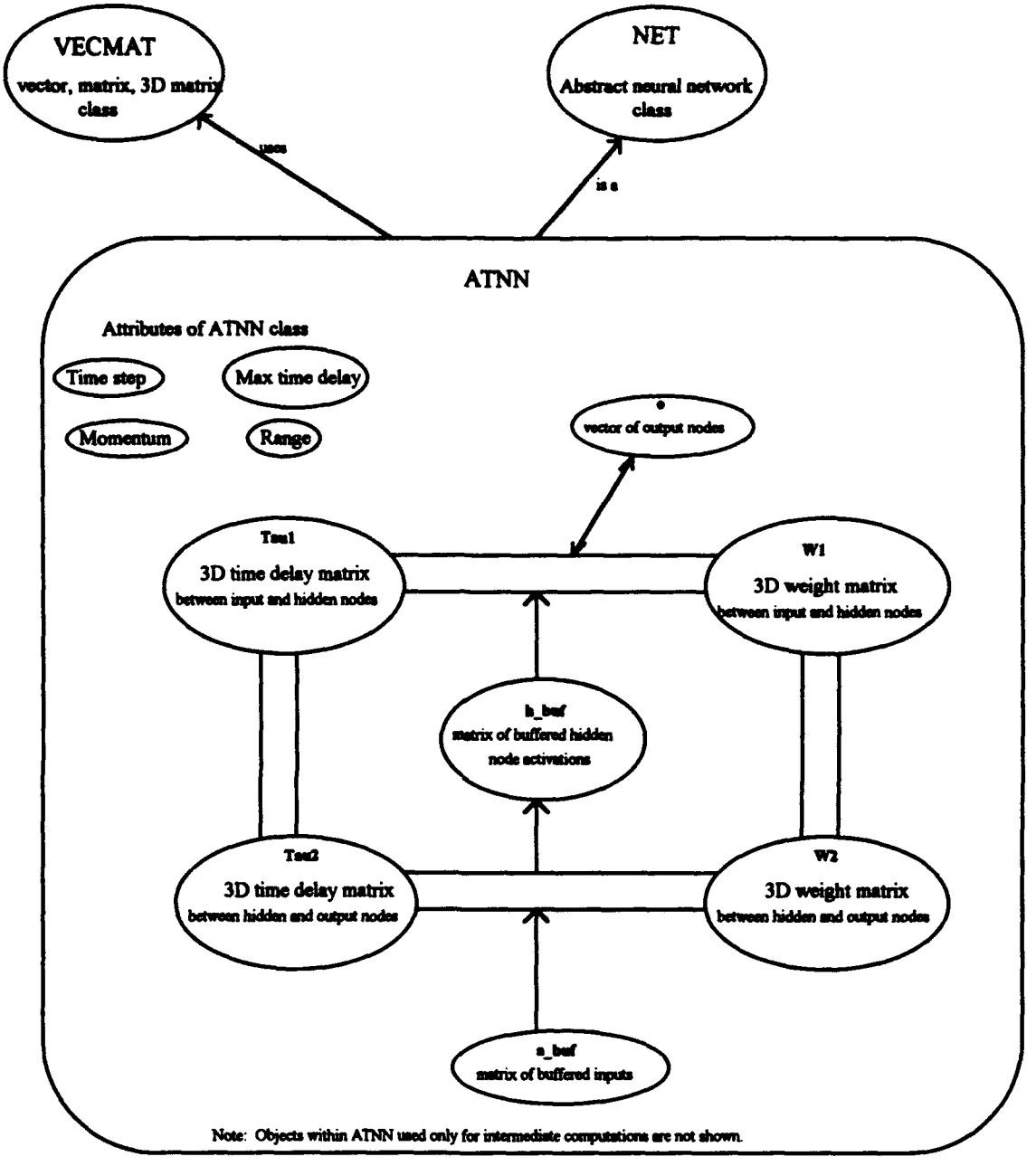


Figure 7: Class hierarchy for ATNN

3.4 *TDNN and BP Implementation*

The Time Delay Neural Network and Error-Backpropagation algorithms can be viewed as special cases of the ATNN algorithm. Fixing the time delays (i.e., not allowing them to update and learn), while still allowing the weights to update, results in a network equivalent to the typical TDNN. By fixing the time delays and setting the maximum number of delays to one (MAXNUMTAU 1 in . DEF file), the Error-Backpropagation algorithm (where $\tau_{jik} = 0$) results. In this ATNN code, the automatically buffered inputs still exist, thus each input feature only requires one input node for the TDNN window of delayed inputs. This thesis uses only the TDNN special case. The network definition file (.DEF file) sets the TDNN condition by simply setting TDNN 1 for the fixed time delays case. If TDNN 0, time delays will update and learn as in the general ATNN case.

3.5 *RTRL Implementation*

This section describes how another type artificial neural network, the subgrouped Real-Time Recurrent Learning (RTRL) algorithm, was implemented for comparison in this thesis. The RECNET code, originally written by Lindsey and, then, modified by Dean, is used much the same as described in Appendix A of Lindsey's thesis [10]. The "parameters.dat" file includes more network definitions, each of which is explained in a comments section of the file itself, due to Dean's modifications. The data file structure remains the same as that in Lindsey's code.

The RECNET code requires a different data format than the ATNN code. The data file structure for using RECNET includes on the top line: the number of input features, number of external output nodes, the total number of nodes (external output plus hidden), and the number of input-output pattern pairs. All subsequent lines contain the input features (# columns of these equates to the first number in the top line) and desired

outputs (# columns of these equals the second number in the top line) all separated only by spaces. Each line corresponds to a separate timed event in the time series process.

RECNET stores its output and error information in a manner such that each different network configuration and data set requires its own directory. Otherwise, the resulting network information from one network overwrites the previous output files.

3.6 Applications

Time series function prediction was attempted for two specific applications. The first application chosen was a sum of incommensurate sine waves. This means the ratio of the sine wave frequencies is an irrational number resulting in a nonperiodic function. In his thesis, Captain James Stright provides a measure of the randomness of this function called the fractal dimension [17]. Fractal dimension, as determined by the Grassberger-Procaccia method used by Stright, begins at 1.0 for a straight line. Very smooth curving data has a low fractal dimension and more "ragged" data has a higher fractal dimension. . For the incommensurate sine wave function defined by

$$y(t) = \left[2 + \sin(\sqrt{2}t) + \sin(\sqrt{3}t)\right]/2, \tag{3.10}$$

the Grassberger-Procaccia method yields a fractal dimension of 1.7. Figure 8 shows a graph of a small sample of $y(t)$ from equation 3.10. A higher sample rate would make this data smoother, however this is the data used for testing and training here. This data should be of low enough dimensionality that artificial neural network prediction is learnable with a reasonable number of nodes and/or time delays. Stright's predictor network obtained an average MSE of 0.349 from the instantaneous MSE of the points given in his thesis. This result though was from a multilayer perceptron implementation

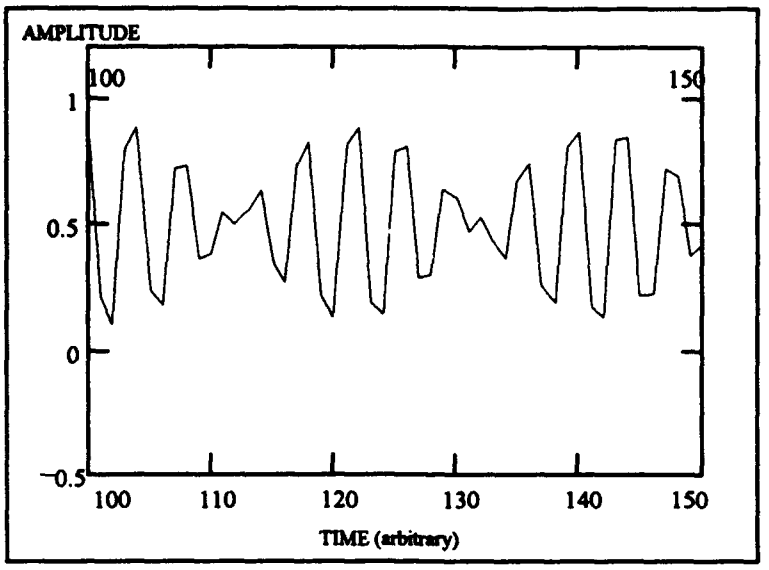


Figure 8: Sum of Two Incommensurate Sine Waves

and was only trained for 1200 epochs with the previous 3 inputs and a total of 20 hidden nodes (15 in one layer and 5 in another). Although this is a benchmark for this application, only the three networks explicitly discussed in this thesis will be compared. Chapter IV discusses the results of this application in terms of the function prediction comparisons.

The second application considered for this thesis contains data even more complex than the incommensurate sine wave data. Data with fractal dimension between 1.95 and 7.5 is considered. Prediction of nonlinear time series functions such as pilot head motion given the time series position (in x, y, z 3D Euclidean distance space) or acceleration data are of particular importance to the Air Force. These type functions probably have an order of dimension near or within that of chaotic data. But if a prediction network predicts future values of chaotic data more accurately than 50 percent of the time, it is beating chance and should perform quite well on head motion data. Since a readily available source of data exists in the area, a set of financial data (British pound opening price data) was chosen as an example data set that is usually considered to be in this category of chaotic data. Figure 9 shows a sample of the data set used for this

application. It is, definitely, a good example of a nonlinear, dynamic process which can not be predicted consistently and accurately by the human brain. The equations for market price processes are currently unknown, but in fact the process is assumed not to be random. The equations are most likely nonlinear, stochastic, delay-differential equations because of market response to certain real-world inputs. [2]

Comparison of similarly configured networks and a discussion of the abilities of each type network to learn these application tasks is presented in Chapter IV.

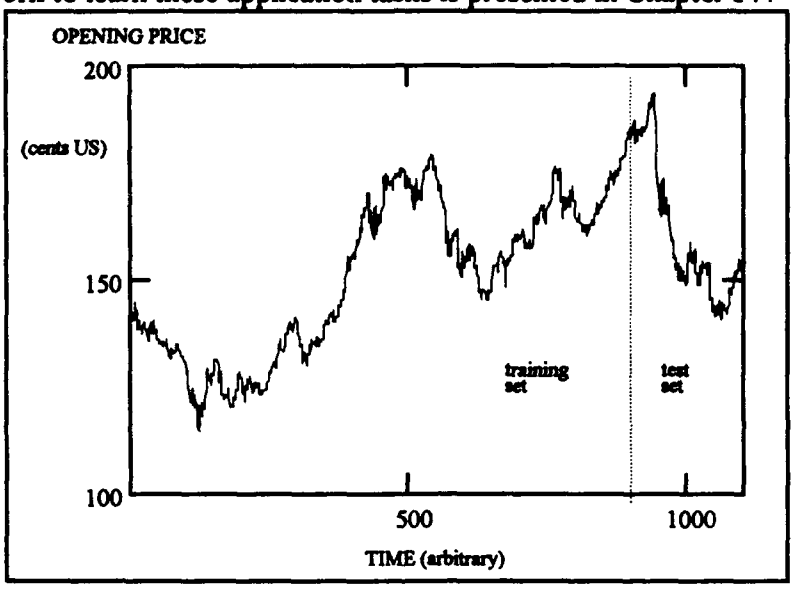


Figure 9: Financial times series data

3.7 Training and Testing the Algorithms

Each of the data sets was scaled by the ATNN code. However, normalizing the data (i.e., zero mean divided by the standard deviation) prior to input allows the networks to learn without saturating the sigmoid functions. Thus, the network prediction should more accurately follow the desired output. Experience shows that prediction of real world processes must be made over very short time frames because the causal forces driving

these processes change so rapidly. For that reason, this thesis only attempts to predict one time step ahead.

Initially, each of the three network types (i.e., RTRL, TDNN, and ATNN) trained on each of the applications. Using one input, one output, various numbers of hidden nodes, and various time delays, a configuration evolved, for each training set, such that the networks learn to predict reasonably well. With these network configurations established, training commenced to a point where the error dropped considerably and then leveled. In the RTRL algorithm, the learning rate automatically decays by a factor of 2 once the MSE levels. Therefore, the ATNN and TDNN training runs were continued in the same manner through the interactive program. Testing was performed at several intermediate points to ensure the networks were indeed learning. From the point where the MSE began to drop significantly, the networks were trained only a few iterations at a time with testing in between the training runs. The goal of this method was to minimize the testing set error while maintaining a low training set error. The result, hopefully, gives an optimized set of weights and/or time delays for each network.

3.8 Summary

This chapter describes the methodology for developing, training and testing the ATNN algorithm and code. It also discusses the implementation of the TDNN (a special case of the ATNN) and the subgrouped RTRL algorithms to be compared with the ATNN. A brief description of the applications used to train and test the networks is followed by the procedures used for collecting comparison data. Chapter IV contains all the comparison results of the applications using the three different types of artificial neural networks as well as a discussion of the prediction capabilities of each.

IV. RESULTS AND DISCUSSION

Chapter III covered the development, training, and testing of the Adaptive Time Delay Neural Network (ATNN), as well as the implementation of the Time Delay Neural Network (TDNN) and subgrouped Real-Time Recurrent Learning (RTRL) algorithms. It discussed a means for comparing these algorithms as applied to the problem of predicting nonlinear, time series processes. This chapter contains the results achieved for each of the three types of artificial neural networks and a discussion which compares their prediction capabilities. Two specific nonlinear time series applications were studied. The first, the sum of two incommensurate sine waves, tackles a relatively easy prediction task as proof that the new ATNN algorithm works reasonably well as a prediction network. The second application, predicting the time series function related to a set of financial data, covers a much more difficult prediction task. Given, as a single time series process, only the historical data associated with a real world problem, predict the future activity of the process. There is more embedded information in such a process than even the human brain can accurately predict with any great consistency. The artificial neural networks, on the other hand, are capable of predicting even the financial time series data to some degree. Thus, if the result can be obtained in time to be useful, the human user of the system will be relieved of some of the burden involved in trying to make sense of the raw data from inadequate or noisy sensing devices.

The following results show that each network is able to predict with a great deal of accuracy. Instead of comparing the networks on the basis of some arbitrary tolerance for determining percent correct, a Mean Square Error (MSE) from Equation 3.2 is used; it relates the target output and the predicted output directly. To set the stage for comparison, a great deal of training and testing runs of each network's code was

performed. This trial and error method was not to optimize the configuration of each type of network individually but, instead, to determine the best configuration which could predict reasonably well with all three. The result provides a means for comparing network performance in terms of training time efficiency and mean square error (MSE) of predicted outputs.

4.1 Incommensurate Sine Wave Results

The sum of incommensurate sine waves exemplifies a function which is just on the verge of non predictability, in human terms. It is a good starting point from which to determine the capabilities of an untested predictor network. For the results presented here for the RTRL, TDNN, and ATNN, each network contained one input node, one output node, and 15 hidden nodes. The task was to predict only to one time step ($t+1$) in the future. The RTRL was given only the time delayed output for the previous input. The TDNN and ATNN inputs were buffered to allow a window size up to the last 20 input time samples (max $\tau_{jik} = 20$).

4.1.1 RTRL RESULTS

The RTRL prediction, as shown in Figures 10 through 12, resulted in a time averaged MSE below 0.001. As seen in Figure 10, the predicted values usually overshoot or undershoot the actual desired output a little but the functional form is maintained almost exactly throughout the test set. For this more simple (low order dimensionality) application, RTRL provided the best prediction results with the given network parameters. The RTRL network started the training initially with a very low MSE, because it was designed to provide feedback and the best possible results in real-time. Thus, Figure 12 shows that the RTRL has a shallow, almost linearly decreasing learning curve.

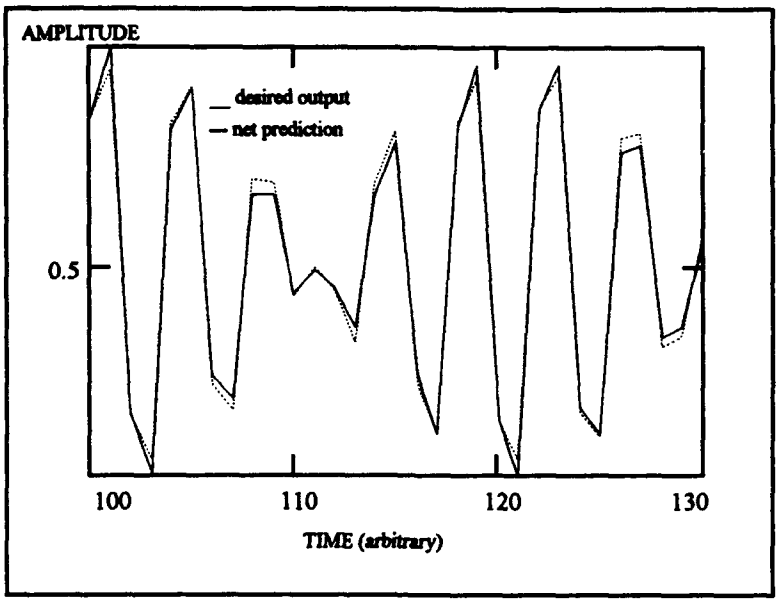


Figure 10: Incommensurate Sine Wave Prediction with RTRL Test data (15 hidden nodes)

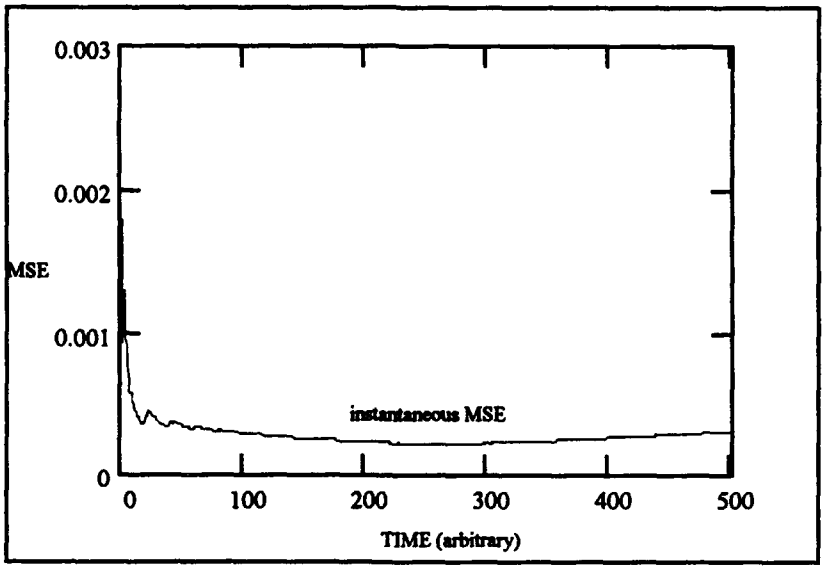


Figure 11: Incommensurate Sine Wave Prediction with RTRL Mean Square Error (MSE) for test data

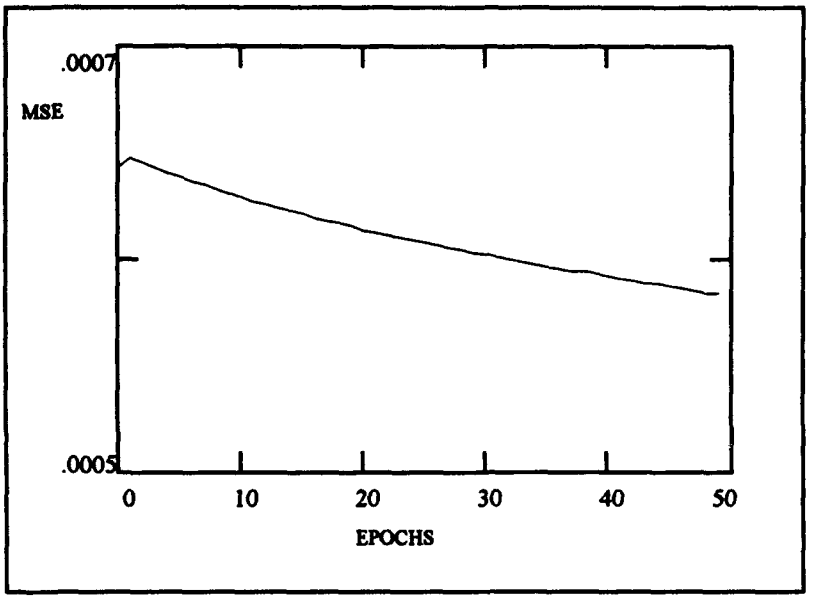


Figure 12: Incommensurate Sine Wave Results with RTRL average Mean Square Error (MSE) during training (15 hidden nodes)

4.1.2 TDNN RESULTS

Predicting the incommensurate sine wave function with fixed time delays, using the TDNN special case of the code developed in this thesis, proved that the algorithm is capable of learning the prediction task with the same number of hidden nodes as the RTRL. With no feedback mechanism, the TDNN must rely on the past inputs, versus previous outputs derived from learning in the RTRL, to learn the appropriate input-output relationships of a function. The prediction results shown in Figure 13, for testing after the first 300 training epochs, were the best obtained for TDNN. More training did not significantly change the MSE as can be seen in Figure 14. The initial learning curve for the TDNN was quite steep and then remained stable. The predicted output maintains the general form of the desired output but the peaks are smoothed instead of sharp decision points. Also, the predicted output appears to lead the desired output in quite a few instances thus increasing the error.

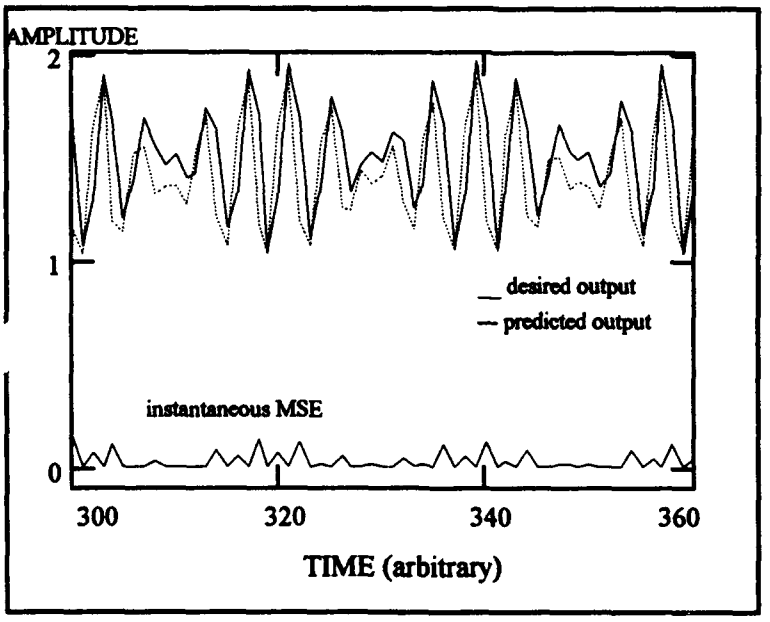


Figure 13: TDNN Incommensurate Sine Wave Prediction Test Data (15 hidden nodes, 10 time delays)

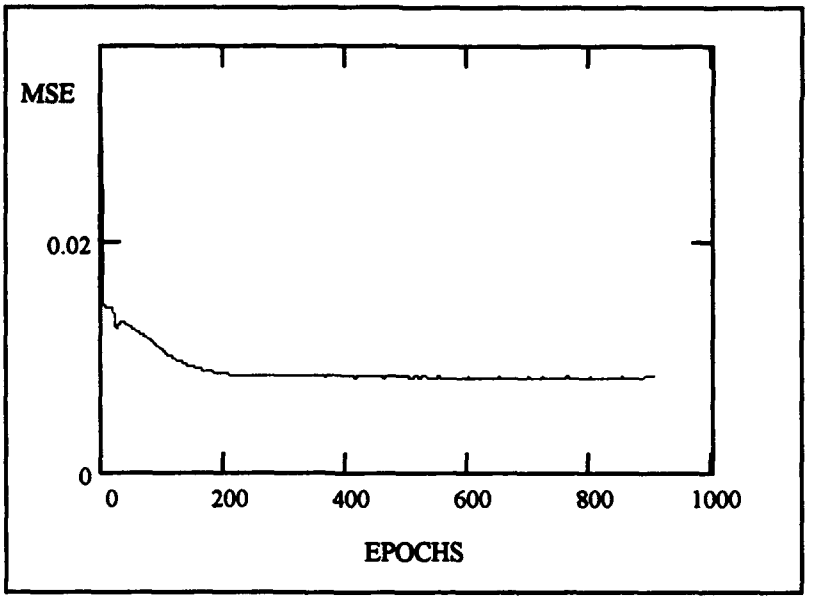


Figure 14: Incommensurate Sine Wave Prediction with TDNN average MSE per epoch during training (15 hidden nodes, 10 time delays)

4.1.3 ATNN RESULTS

In applying the ATNN algorithm to the incommensurate sine wave prediction task, it was hoped that a better time dependency relationship could be learned. Figure 15 shows that the ATNN was unable to significantly improve upon the learned time relationship found with the TDNN. There is an improvement, however, in that the ATNN relates the peaks better than the TDNN thus not smoothing the prediction at the peaks. The functional form of the prediction compares more closely to the desired form, but the time averaged MSE was the same as the TDNN for this best case ATNN prediction after 300 epochs. The learning curve presented in Figure 16 shows that the ATNN learns as well as, but in significantly fewer training cycles than, the TDNN. The instability in the learning curve is due to a high learning rate set for the time delay update rule. A decaying time delay learning rate was incorporated manually through the interactive capability of the ATNN code to overcome this instability. Decaying learning rates were proven by Lindsey for the RTRL and were verified in this research to benefit this ATNN algorithm as well. The next application tasks the prediction capability of these algorithms given a real world problem with no accurately known mathematical model.

4.2 Daily Financial Data Results

Financial time series data, as seen earlier, provides an interesting example of a real world dynamic process about which very little is known as far as rigorous mathematical modeling goes. Historical data is readily available. British Pound opening price data was chosen for this application. This example has proven in the past to be highly unpredictable by human forecasters tracking raw data. It is hypothesized that a neural network which incorporates time dependency relationships will aid the human in assimilating the information, as normally provided, to better predict future values.

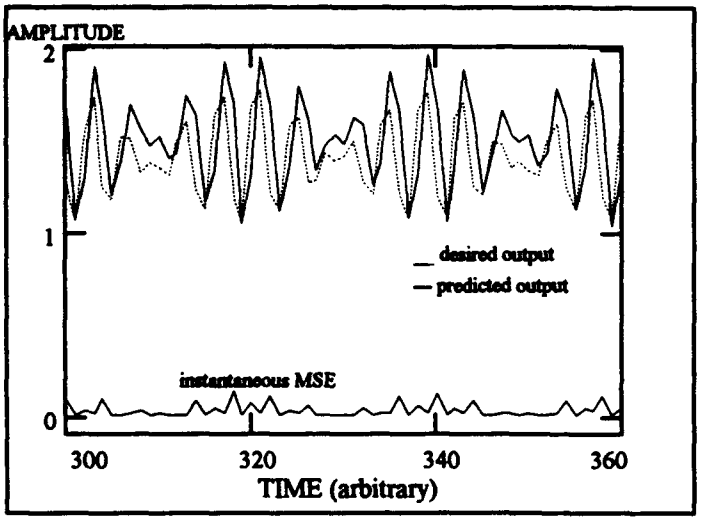


Figure 15: Incommensurate Sine Wave Prediction using ATNN a sample of test data (15 hidden nodes, 20 time delays)

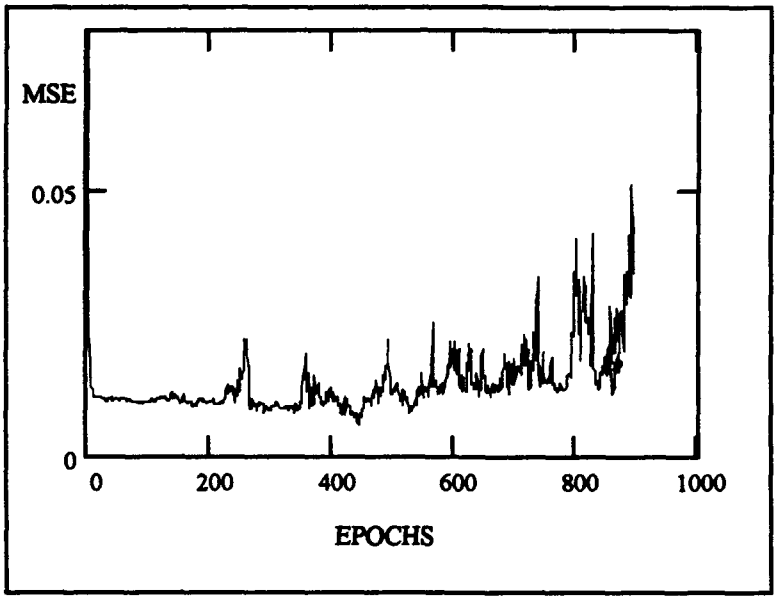


Figure 16: Incommensurate Sine Wave Prediction using ATNN time averaged MSE per epoch for training data (15 hiiden nodes, 20 time delays)

Again each of the three networks was trained and tested as discussed in Chapter III. In all the results presented here, it was found that the best configuration for comparing the networks, given one input and one output, would contain 15 hidden nodes. The TDNN and ATNN were provided with a maximum of 20 time delays. The case of providing only 10 time delays to these two networks is also presented for comparison. The task was still to predict only to one time step ($t+1$) in the future.

4.2.1 RTRL RESULTS

Applying the RTRL algorithm to the given data proved unsuccessful in predicting the desired output because the test data, although normalized on the entire data set, contained very few points that related directly to the training data set provided. Better results were obtained after scaling the input data in the same manner as the ATNN code performs automatically. This extra procedure prevents the saturation of the sigmoid function so that when the test data lies outside the norm, usable results can still be obtained. Results of the RTRL are presented in Figure 17 through 19. As seen in Figure 17, the RTRL does learn the financial data time series quite well. However, Figure 18 shows that as this network predicts on points farther away from the actual known data, the instantaneous MSE gradually increases. Since the RTRL feeds the output, with its associated error, back to the input and has no long term memory, this was expected. Figure 19 shows the resulting average MSE, or learning curve, during training. As seen, the RTRL learns to predict with a time averaged MSE to about 0.001. This RTRL prediction capability will be compared to that of the TDNN and ATNN in the next subsections.

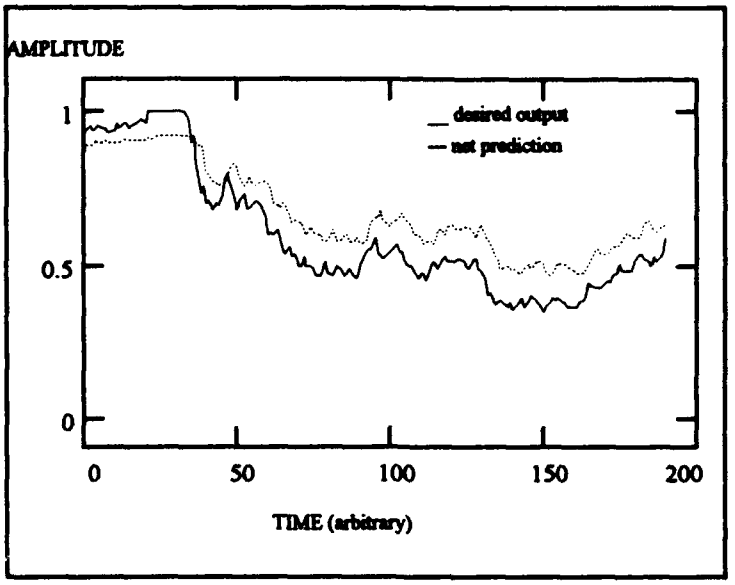


Figure 17: Financial Test Data Prediction with RTRL (15 hidden nodes)

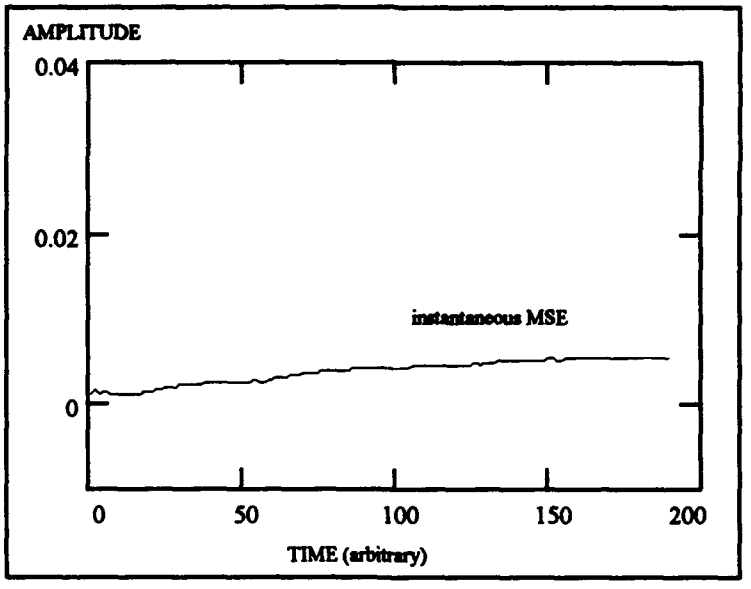


Figure 18: Financial Test Data Prediction with RTRL instantaneous Mean Square Error (MSE) for test data

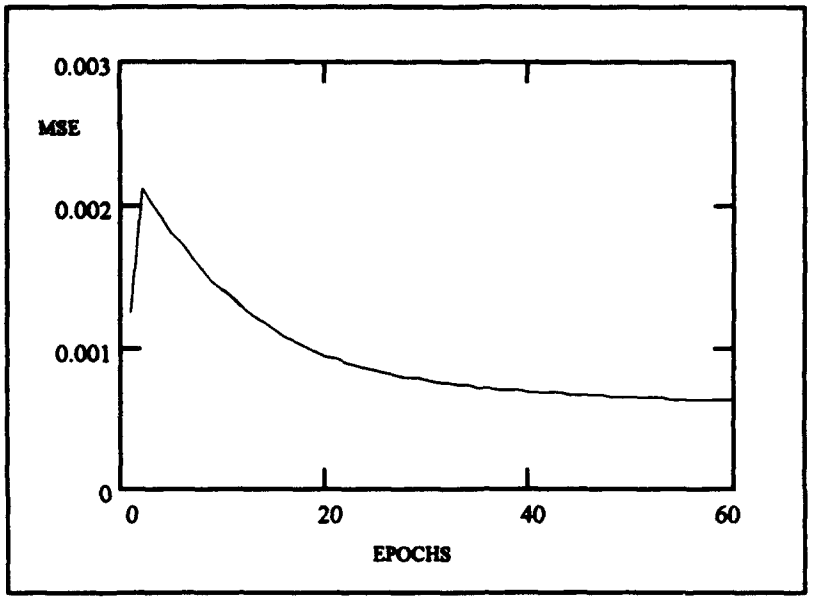


Figure 19: Financial Training Data using RTRL
average Mean Square Error (MSE) per epoch

4.2.2 TDNN RESULTS

Results of predicting the next value of the British pound data, using the TDNN with 10 time delays, are shown in Figure 20. Comparing this to a TDNN with 20 time delays, as in Figure 22, shows the importance of adding an acceptable amount of memory capability. The 20 TDNN results follow the functional form of the desired output much more closely than with just 10 time delays. These prediction results are given for the test data after training the 10 and 20 time delay networks for 1000 and 500 epochs, respectively. These were the best test results obtained over the whole training period. The time averaged MSE for the 20 delay case proves only slightly better than with 10 delays, as shown in Figures 21 (for 10 time delays) and Figure 23 (for 20 time delays). This is true because the 10 delay case has a portion of the results right through part of the test set, but even there it does not follow the time varying nature of the real process. With fewer training cycles, the 20 delay case clearly attempts to follow the real process the best.

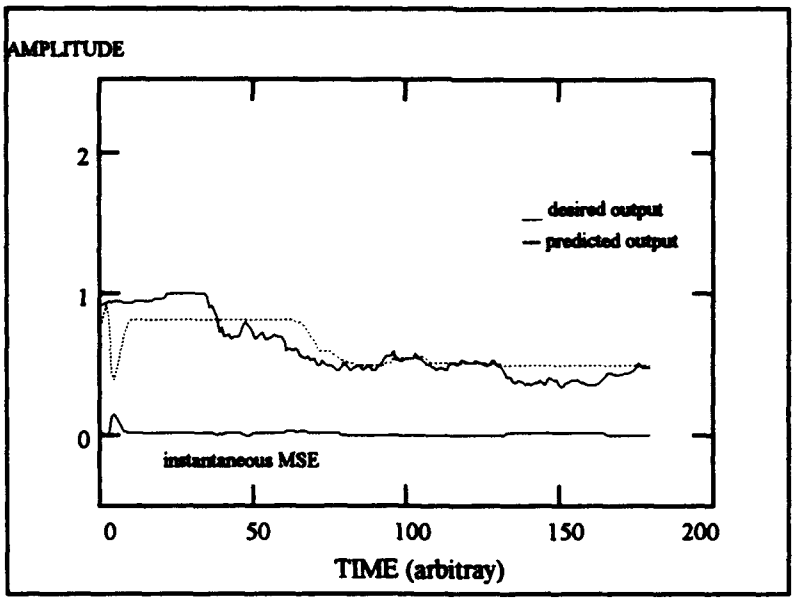


Figure 20: British Pound Data Prediction with TDNN Test data (15 hidden nodes, 10 time delays)

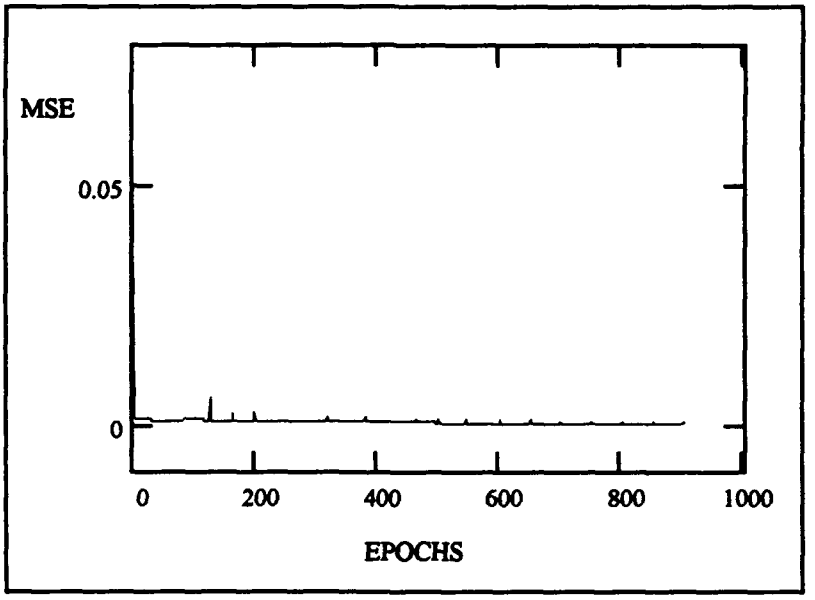


Figure 21: British Pound Data Prediction with TDNN average MSE per epoch (15 hidden nodes, 10 time delays)

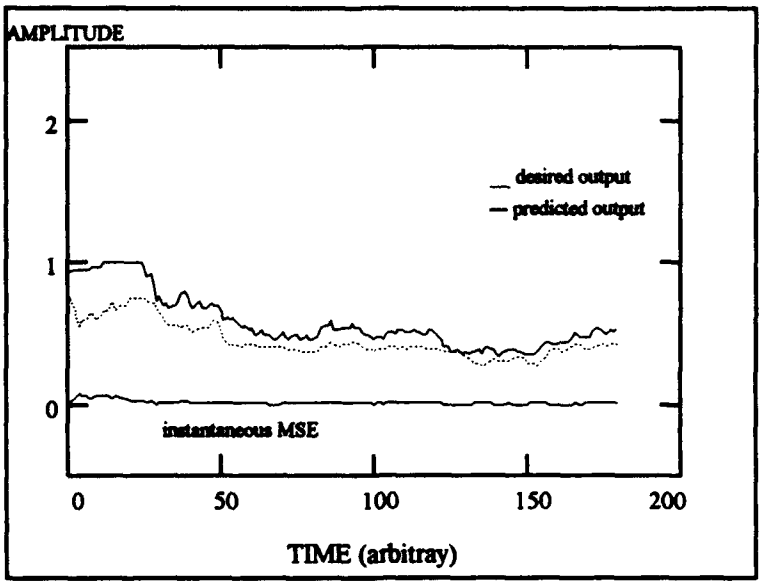


Figure 22: British Pound Data Prediction with TDNN Test data (15 hidden nodes, 20 time delays)

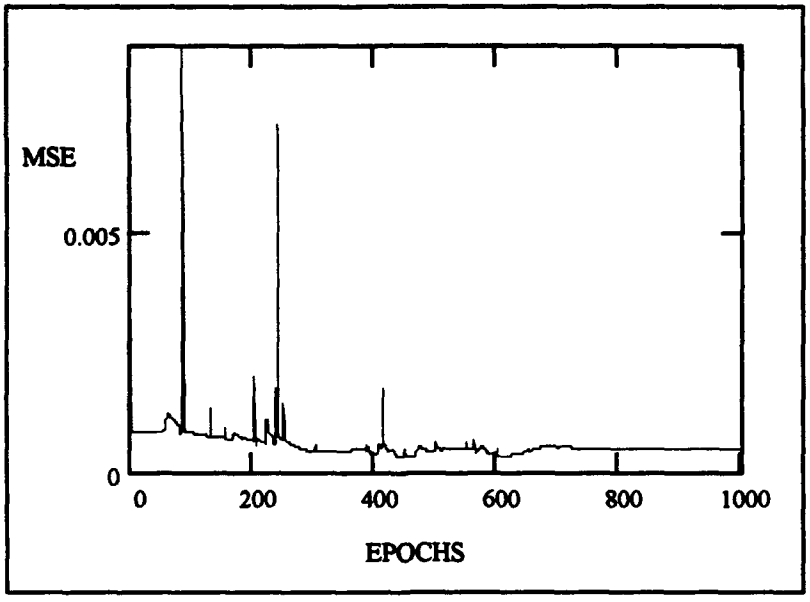


Figure 23: British Pound Data Prediction with TDNN average MSE per epoch (15 hidden nodes, 20 time delays)

It should be noted also that the first $2K$ data points in each test set should be the last $2K$ known data points of the process (where K is the maximum number of time delays as configured in the .DEF file). This is because the hidden node buffer must be filled before prediction actually begins, thus providing a longer term memory for TDNNs than RTRL.

4.2.3 ATNN RESULTS

Applying the ATNN resulted in more erratic learning when high fixed learning rates were used as shown in Figure 25, but the predicted values match the functional form of the test data set even better than the best TDNN case or the best RTRL. Although the predicted values are slightly different from the desired output, they follow the peaks of the real process with considerably more detail than the TDNN (see Figure 24 for the 10 time delay ATNN). The minimum MSE is an order of magnitude better than either the RTRL or the TDNN case. Again, as with the TDNN, the first $2K$ values of the test set must be the known past time samples. The increased instantaneous MSE in Figure 26 is the result of not allowing the buffer to fill properly. The predicted values for this network do follow the desired output but not as well as for the 10 delay case. Figure 27 shows the results of the same 20 delay network (after 745 epochs) when the buffers are allowed to fill prior to the desired prediction. Better instantaneous MSE is obtained for the actual test prediction points. As seen in Figure 28, a more stable averaged MSE results during training for this network. The learning rates were held at 0.1 for this network during training. Thus, small learning rates keep the network more stable but require more training epochs to obtain even the accuracy seen in Figure 27. Here again, it is seen that the amount of memory required to learn a particular process is important. Too much memory can "confuse" the network requiring extra long training times. The ATNN does prove capable of learning the time relationships better, as in the 10 delay case, by an order of magnitude than the other two networks tested.

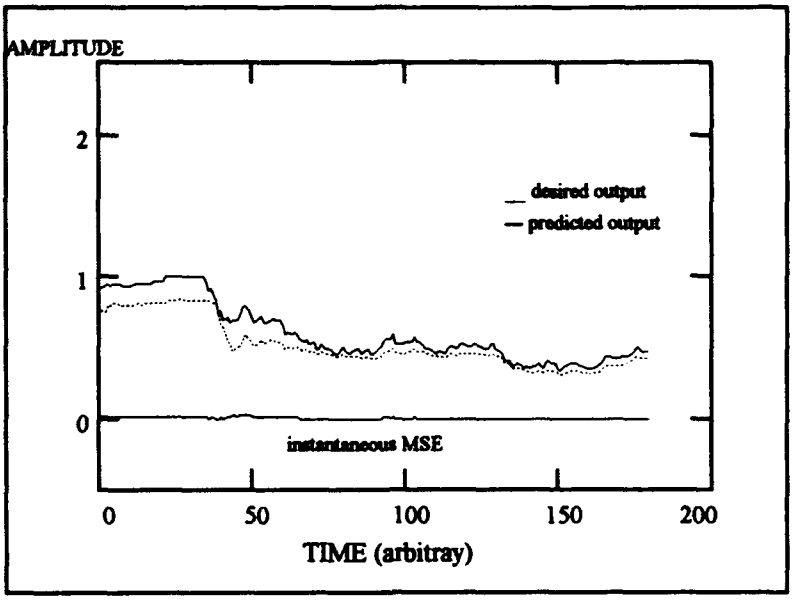


Figure 24: British Pound Data Prediction using ATNN Test data (15 hidden nodes, 10 time delays)

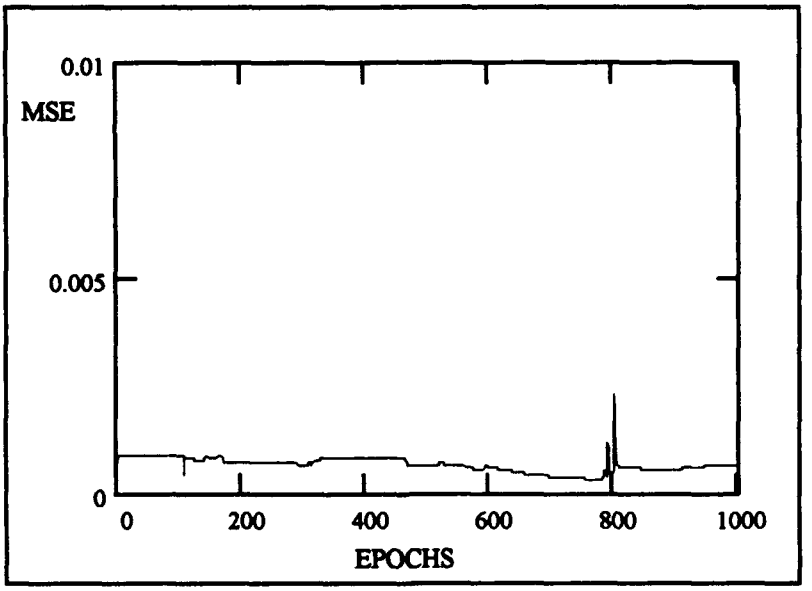


Figure 25: British Pound Data Prediction using ATNN time averaged MSE per epoch (15 hidden nodes, 10 time delays)

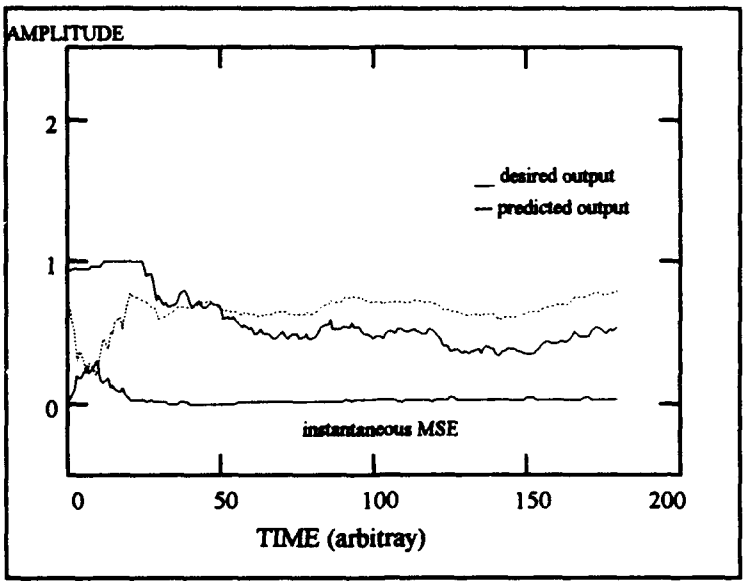


Figure 26: British Pound Data Prediction using ATNN Test data (15 hidden nodes, 20 time delays)

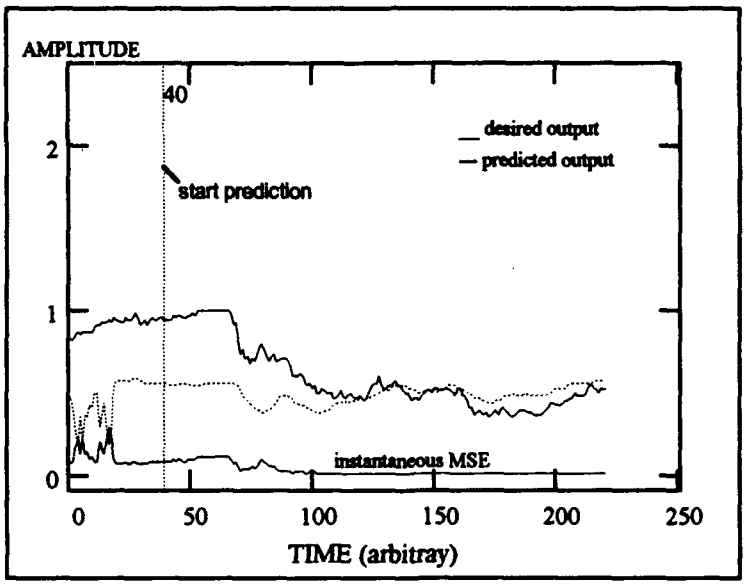


Figure 27: British Pound Data Prediction using ATNN Test data (15 hidden nodes, 20 time delays)

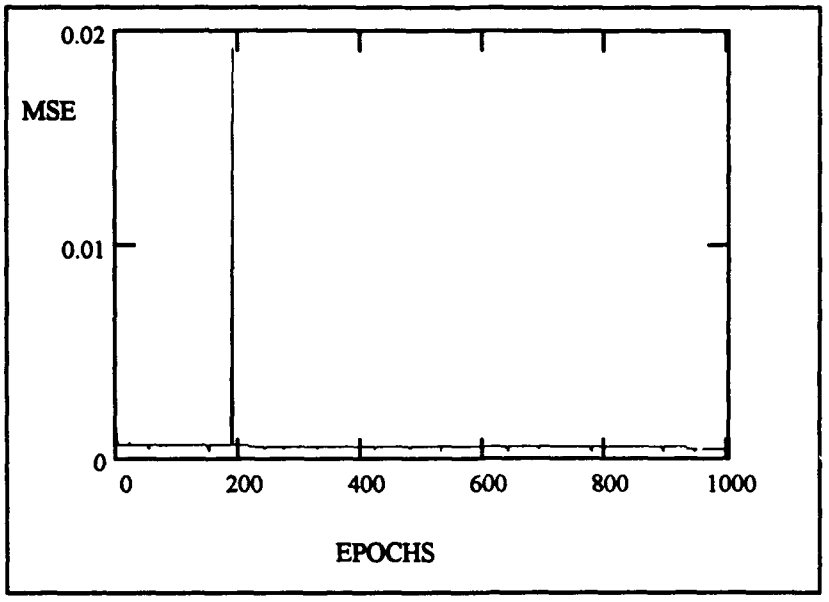


Figure 28: British Pound Data Prediction using ATNN time averaged MSE per epoch (15 hidden nodes, 20 time delays)

4.3 Discussion

The results presented in the previous section clearly show that for a more simple function, the subgrouped RTRL algorithm performance is almost unbeatable. The TDNN and ATNN predict the incommensurate sine wave fairly well compared to the RTRL. The MSE for the RTRL is an order of magnitude better on the incommensurate sine data than that of time delay networks. The RTRL algorithm, even this 10 times faster subgrouped RTRL, takes 12 times longer per training cycle than the ATNN code. Running on the 100Mhz Silicon Graphics ONYX systems, the RTRL implementation utilizes only a single processor per neural network. The object-oriented programming design of the ATNN allows the work to spread over the available processors (there are 4 running at 100Mhz each on the ONYX). Once trained, though, all three type networks predict in roughly the same amount of time as would be expected since the number of hidden nodes was held the same and since the networks are no longer learning. As seen in Figures 11and 18, the instantaneous error begins to increase slightly as the RTRL prediction gets

farther from the known values. With the amount of time it takes to train this RTRL algorithm may be impractical for some real world problems. Retraining is, probably, necessary quite often to accurately predict highly dynamic processes. The RTRL may have to be retrained even more often than feedforward type networks since error, which feeds back to the input in the RTRL, may eventually degrade its abilities. The time delay type networks may be better for predicting real world processes, which are often more nonlinear and more complex, thus requiring some amount of long term memory incorporated in the learning process.

The results, for an example of this real world type process, show that the ATNN code developed here will be more adaptable to the inputs especially if their values are not known to directly relate to the trained values before testing. Once the input data from past known events are buffered the ATNN learns to predict the direction of change for the future values of a real world process exceptionally well even for test data that is outside the norm of the training set. Therefore, for predicting the direction of future values of a highly nonlinear process, the ATNN wins, by an order of magnitude in MSE, over the RTRL or TDNN.

4.4 Summary

This chapter discusses the results of research performed using the ATNN code developed in this thesis. By comparing the ATNN and TDNN, the importance of learning the best time delay values became apparent. By varying the fixed time delays in the TDNN as well as letting the ATNN do that work, better prediction resulted. Two specific nonlinear time series applications were studied. The RTRL beat the ATNN and TDNN for the more simple nonlinear function in terms of absolute accuracy. However, The ATNN learned quickly and provided very accurate prediction of the process direction. The ATNN outperformed the others by far when given a complex, real world processes.

After presenting the results, a brief discussion is included on the computation times of the computer codes used (i.e., ATNN and RECNET). The algorithm for ATNN inherently takes less time than RECNET. RECNET also suffers in that it only runs on a single processor whereas ATNN makes use of the multi-processor environment when available. Conclusions and recommendations for future work in this area are given in the next chapter.

V. CONCLUSIONS AND RECOMMENDATIONS

This thesis implements an Adaptive Time Delay Neural Network (ATNN) capable of user-defined degeneration to the more common Time Delay Neural Network (TDNN) or Error-Backpropagation Network (BP). The algorithm test results and time series function prediction capabilities as compared to the RTRL algorithm show the advantages and disadvantages of ATNNs for prediction. Time series prediction, defined as determining future value of a process based on historical data, applies to a great number of real world situations. Many of these applications are, also, extremely important to the Air Force.

5.1 Conclusions

Lindsey demonstrated the RTRL algorithm's ability to learn several time dependent functions. He showed that the RTRL was more robust than the best linear predictor. So this thesis compares an RTRL algorithm to the prediction ability of a new algorithm, ATNN, which learns the optimum time delays on the input facts. This new approach to solving the time series function prediction task proves useful in both determining the direction of future values and taking less computation time during training. Although the RTRL clearly beat the ATNN at predicting for a known predictable function, the ATNN still performed to within 10 percent of the almost exact RTRL capability. The TDNN prediction capability came in last for this case because a smoothing effect at the nonlinearities. It was the ATNN that bested both the TDNN and RTRL when applied to a real world process with no known mathematical function. The RTRL had problems handling different inputs than those of its training set due to a scaling problem. The TDNN again learned the overall trend in the data set but tended to smooth over the nonlinearities. Thus for a highly nonlinear process the ATNN outperformed the others by

far. This comparison testing demonstrates that the ATNN should be very good at predicting any process, but if the process has a high degree of randomness, the ATNN can pick out and predict even the fine sample to sample relationships. The other networks tested here could not.

Every algorithm studied as part of this research effort incorporates its own strengths and limitations. What works best will depend on the problem the network attempts to solve. There exist many ways to tailor a network. Fit between the problem and solution governs the choices. It seems evolution may try various modifications in network design, use them for a while, then keep those that solve the problems the best. The central nervous system does not conform to a single network layout but uses the right design in the right places. The best network design depends on the task and how the evolutionary choosing process shook out. For engineers trying to match artificial neural networks to problems and solutions, the most effective approach might be to develop a toolbox containing various network layouts, apply those that make sense, and keep only the most efficient network for the given problem. This thesis steps toward the process of building an effective toolbox for solving real world problems.

5.2 Recommendations

Real world inputs generally have more dimensions than those chosen in a laboratory. This scaling problem concerns whether the scaled network can incorporate all the relevant dimensions and still perform the prediction task in real time. Also, how are the correct inputs to be add to the relevant dimension of a complex process determined? The ATNN code developed for this thesis allows the user to provide a large number of input features than the typical TDNN input scheme for trying to predict the future values in a given

process. Therefore future work could incorporate this ability by providing the network with other processes that seem to effect the process under study.

Many processes associated with the real world are highly nonlinear and seemingly unpredictable. These are very interesting to study because these "random" processes usually are not completely random. Potential exists for moving them from the realm of unpredictable to some degree of predictable using artificial neural network technology. One very interesting problem for the Air Force is predicting pilot head motion, which is often a highly nonlinear process. Prediction seems possible given the pilot's situation at a particular instant in time is known. This prediction could be used to update virtual cockpit displays faster. By predicting where the head will be just a tenth of a second ahead, it is possible to make the computer generated scene on a helmet mounted display seem more real, or virtual. This thesis provides a comparison of one method, ATNN, to another, RTRL, which might be used to perform this prediction task. Much more research is needed in the area of neural networks, and research using them for predicting time series processes has really just begun.

APPENDIX A. Time Delay Update Rule

This appendix presents the derivation of the time delay update rule found in the article by Lin [9] and used in this thesis.

Define an instantaneous error measure (MSE) as in Equation 3.2.

$$E(t_n) = \frac{1}{2}\sum_{j \in P}(d_j(t_n) - a_j(t_n))^2$$

where P is the number of output nodes with computed values, $a_j(t_n)$. The term $d_j(t_n)$ denotes the desired output of node j at time t_n. Then by the chain rule

$$\frac{\partial E(t_n)}{\partial \tau_{jik}} = \frac{\partial E(t_n)}{\partial S_j}\frac{\partial S_j(t_n)}{\partial \tau_{jik}} \tag{A.1}$$

The second term in Equation A.1 is given by

$$\begin{aligned}\frac{\partial S_j(t_n)}{\partial \tau_{jik}} &= \frac{\partial}{\partial \tau_{jik}}\sum_{i=0}^{N}\sum_{k=1}^{K} w_{jik}a_i(t_n - \tau_{jik}) \\ &= -w_{jik}a_i'(t_n - \tau_{jik})\end{aligned} \tag{A.2}$$

where N is the number of nodes in the previous layer and K is the maximum number of time delays. Now define

$$\rho_j(t_n) = \frac{\partial E(t_n)}{\partial S_j} \tag{A.3}$$

Substitute Equation (A.2) and (A.3) into Equation (A.1), to obtain

$$\frac{\partial E(t_n)}{\partial \tau_{jik}} = -\rho_j(t_n)w_{jik}a_i'(t_n - \tau_{jik}) \tag{A.4}$$

Thus the learning rule as in Equation (3.8) is obtained

$$\Delta\tau_{jik} = \eta_1 \rho_j(t_n) w_{jik} a_i'(t_n - \tau_{jik}) \tag{A.5}$$

To derive $\rho_j(t_n)$, apply the chain rule and consider two cases:

$$\begin{aligned}\rho_j(t_n) &= \frac{\partial E(t_n)}{\partial S_j} \\ &= \frac{\partial E(t_n)}{\partial a_j}\frac{\partial a_j(t_n)}{\partial S_j} \\ &= \frac{\partial E(t_n)}{\partial a_j} f'(S_j(t_n))\end{aligned} \tag{A.6}$$

To find $\frac{\partial E(t_n)}{\partial a_j}$, consider two cases:

1. If j is an output node:

$$\frac{\partial E(t_n)}{\partial a_j} = -(d_j(t_n) - a_j(t_n)) \tag{A.7}$$

and thus Equation (A.6) becomes

$$\rho_j(t_n) = -(d_j(t_n) - a_j(t_n)) f'(S_j(t_n)) \tag{A.8}$$

2. If j is a hidden node:

$$\begin{aligned}\frac{\partial E(t_n)}{\partial a_j} &= \sum_{p \in P} \frac{\partial E(t_n)}{\partial S_j}\frac{\partial S_j(t_n)}{\partial a_j} \\ &= \sum_{p \in P} \frac{\partial E(t_n)}{\partial S_j}\frac{\partial}{\partial a_j}\left(\sum_{i \in N}\sum_{q=1}^{K} w_{jik} a_i(t_n - \tau_{jik})\right) \\ &= -\sum_{p \in P} \rho_p(t_n)\left(\sum_{q=1}^{K} w_{pjq}\right)\end{aligned} \tag{A.9}$$

where P is the number of nodes in the next layer, N is the number of nodes in the previous layer, and K is the maximum number of time dealys. For this case Equation (A.6) becomes

$$\rho_j(t_n) = -[\sum_{p \in P} \sum_{q=1}^{K} \rho_p(t_n) w_{pjq}(t_n)] f'(S_j(t_n)) \qquad \text{(A.10)}$$

APPENDIX B. Using the ATNN Code

B.1 An Interactive Environment

A main program, named testatnn, controls the operation of the ATNN algorithm. It provides an interactive environment for training (the default option), testing, and running the ATNN using the "**-L**", "**-T**", "**-R**" options, respectively. The **atnn** command, if given alone, defaults to a training example of the network named ATNN which requires a definition file (ATNN.DEF) and a fact file (ATNN.FCT). The "**-n netname**" option, if added to the command line before the other options, allows the user to specify a particular network name using the structure (in netname.DEF file) for an associated fact file (netname.FCT) to begin training. Another command line option (-v) initiates a verbose mode for tracing the network through the training or testing phase then saving the weights and time delays into a readable ASCII file (located in a netname.MAT file). As an example, the command

atnn -n mytestfile -T -v

will execute the test part of the ATNN algorithm, with verbose output to the monitor, using "mytestfile.TST" as the input fact file.

The program is interactive in that it allows the user to set a maximum number of epochs or target minus network output tolerance for determining program completion, but the training process may be suspended (by pressing the ESC key on the Silicon Graphics or any key on a PC) and saved at any point. Subsequent training automatically resumes from the stored network information (located in the associated .WTS file). This allows for testing at different points during the training process which helps in optimizing training.

Blum uses this same file extension naming convention throughout his neural network tool kit. Implementations require basically the same set of files: files for network definition (.DEF files), training data (.FCT files), test data (.TST files), and run

data (.IN files). Outputs save to a common set of files: binary learned network information (.WTS files), ASCII learned network information (.MAT files) when -v invoked, and run results (.OUT files). Two additional files, implemented for this thesis, save intermediate error information for later analysis and comparison: training error (.ERR files) and test error (.TER files).

B.2 Training with the ATNN

Given the required files, **atnn** configures, dynamically, to the defined network parameters. Memory gets allocated for all the variables during initialization. The program reads each vector pair, one at a time, to the end of the file. Thus, any number of patterns may be presented to the network from the fact file. The network begins learning to optimize the weights and time delays, until either all the facts are within tolerance, the maximum number of iterations, or cycles, is reached, or training is suspended.

During training, ATNN outputs information to the monitor for determining training status. In the default mode, the cycle (or epoch) number displays followed by an "x", indicating a bad guess (outside tolerance), or a ".", indicating a good guess (all outputs within tolerance). In the verbose mode, several messages display information to trace the outputs through the learning phase: Unsquashed guess - the linear output before applying the sigmoid, the Output layer threshold - used in the sigmoid function, Desired outputs, and network Guessed outputs. Also, the words Bad guess or Good guess replace each "x" or "." as appropriate. At the end of each cycle, the average MSE and percentage correct display.

The fact file (with a .FCT extension) contains all the input-output pattern pairs, called vector pairs, for training. In each of these files, the first line defines the vector of minimums for the input features, followed by a comma, followed by the vector of minimums for all the output factors (thus the name vector pair). The second line contains

the maximums for the inputs and outputs in the same manner. The third line is a comment line which begins with a colon (" : "). The subsequent lines each contain the actual vector pairs. The maximums and minimums scale the values of each piece of data to a number between 0 and 1. Values below the specified minimum or above the maximum for each fact get converted to 0 or 1, respectively. Upon training completion, or suspension, the learned parameters get saved to a binary file (netname.WTS file) and computed error and accuracy, for each epoch, go to another file (netname.ERR). The resulting network may then be tested or run.

B.3. Testing and Running the ATNN

When invoked with the **atnn -T** command option, testatnn tests the trained network on the facts stored in the test file (.TST file). The test fact file, formatted the same as the training fact file, contains vector pairs for testing. An output file (with a .TER extension) saves the desired output, network output, and instantaneous Mean Square Error (MSE) for each input vector as well as a time averaged MSE for the entire test data set. To run the trained network, giving it only the input vectors, invoke the **atnn -R** command option. This requires an input file (with a .IN extension) in much the same format of minimum line, maximum line, comment, and input features. However, no desired output vector is included in the input file. The program creates a file of outputs (with a .OUT extension) corresponding to the input features.

APPENDIX C. Adaptive Time Delay Neural Network Source Code

This appendix contains the source code listings for the Adaptive Time Delay Neural Network algorithm developed at AFIT called **ATNN**. It was written in C++ object-oriented programming style and successfully compiles on the Silicon Graphics workstations as well as on an IBM/compatible 486 personal computer using *Turbo C++ 3.0*. The main shell program is named **testatnn.cc**.

```
//////////////////////
// TESTATNN.CC
// Interactive ATNN System Demonstration Program
// Used to verify ATNN system algorithms
// Developed with Turbo C++ 3.0
// Author:  Capt James Gainey, GEO-93D      Last Modified:  10 Sep 93


#define NDEBUG 1  // ANSI method to enable or disable debugging
#include"atnn.hpp"

#include <getopt.h>


char netname[16]="ATNN"; // file where test data is stored
char mode=0; //default to learn or training mode
int trace=0; // SET TRACE=<whatever> at DOS prompt to turn trace on
char *p;

main(int argc,char **argv)
{

  #ifdef __ZTC__
  // shouldn't have to do this! these should be defaults
  cout.setf(cout.unitbuf); // turn "unitbuf" on to force flushing after each char
  cout.unsetf(cout.scientific); // turn skipws and scientific off
  #endif
  cout.precision(2);

  cout << "TEST_ATNN - Interactive Adaptive Time-Delay Network Tester\n";

  int option;

  while ( ( option = getopt( argc, argv, "n:LTRv" ) ) != -1 )
  {
 switch ( option )
 {
        case 'n':
                strcpy( netname, optarg );
                break;
        case 'L':
                mode = 'L';
                break;
        case 'T':
```

```
                mode = 'T';
                break;
        case 'R':
                mode = 'R';
                break;
        case 'v':
                trace = TRUE;
                break;
        default:
                break;
}
}

atnn b(netname);

switch(mode){
case 'T':
  b.test();
  break;
case 'R':
  b.run();
  break;
default:
case 'L':
  b.train();
  break;
```

```
    }

    return 1;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
// ATNN.CC
// Implementation of an Adaptive Time-Delay Neural Net
// Developed with Turbo C++ 3.0
// Author:  Capt James Gainey, GEO-93D        Last Modified:  10 Oct 93


#include "atnn.hpp"

extern int trace;


atnn::atnn(char *s):net(s) // constructor
{
  const NOPARMS = 9;

  PARM parms[NOPARMS];

  strcpy(parms[0].name, "HIDDEN");
  parms[0].type = integer;

  strcpy(parms[1].name, "MOMENTUM");
  parms[1].type = real;

  strcpy(parms[2].name, "INITRANGE");
  parms[2].type = real;

  strcpy(parms[3].name, "MAXNUMTAU");
  parms[3].type = integer;

  strcpy(parms[4].name, "EPOCH");
  parms[4].type = integer;

  strcpy(parms[5].name, "TOLERANCE");
  parms[5].type = real;

  strcpy(parms[6].name, "RATE2");
  parms[6].type = real;

  strcpy(parms[7].name, "TSTEP");
  parms[7].type = real;

  strcpy(parms[8].name, "TDNN");
  parms[8].type = integer;
```

```
readparms(NOPARMS,parms,name);
q          = parms[0].val.i;
momentum   = parms[1].val.f;
initrange  = parms[2].val.f;
K          = parms[3].val.i;
epoch      = parms[4].val.i;
tolerance  = parms[5].val.f;
learnrate2 = parms[6].val.f;
tstep      = parms[7].val.f;
TDNN       = parms[8].val.i;

// initialize weight matrices to random values from -1 to +1
// and time delay matrices to integer values from 0 to K

W1  =new mtrx3d(q,n,K,-initrange);
W2  =new mtrx3d(p,q,K,-initrange);
Tau1=new mtrx3d(q,n,K,K);
Tau2=new mtrx3d(p,q,K,K);

dW1  =new mtrx3d(q,n,K);
dW2  =new mtrx3d(p,q,K);
dTau1=new mtrx3d(q,n,K);
dTau2=new mtrx3d(p,q,K);

a_buf   =new matrix(n,K);
h_buf   =new matrix(q,K);
a_buf_p  =new matrix(n,K);
h_buf_p  =new matrix(q,K);

h=new vec(q);
o=new vec(p);
d=new vec(p);
e=new vec(q);

thresh1=new vec(q);
thresh1->randomize(initrange);
thresh2=new vec(p);
thresh2->randomize(initrange);

if(epoch){
  totd=new vec(p);
  tote=new vec(q);
}

minvecs=new vecpair(n,p);
```

```
  maxvecs=new vecpair(n,p);


  cycleno=0;
}

atnn::~atnn()
{
  delete W1;
  delete W2;
  delete Tau1;
  delete Tau2;

  delete dW1;
  delete dW2;
  delete dTau1;
  delete dTau2;

   delete h;
  delete o;
  delete d;
  delete e;

  if(epoch){
    delete totd;
    delete tote;
   }

  delete minvecs;
  delete maxvecs;

}

//////////////////////////////////////////////////////////
//
// ADAPTIVE TIME DELAY ALGORITHM METHODS - ENCODE AND
// RECALL
//

int atnn::encode( vecpair &v )
{
  float maxdiff;

// Step 1): Propagate through to hidden layer nodes
```

```
// Buffer inputs to a_buf[i][tn] for the last K timesteps

buffer( a_buf, v.a );


// Sum the K buffered time-delay connections. Each connection
// is the product of a time delayed input and the weight
// matrix. Get tdblocks for each input.

propagate(*h, *W1,*Tau1,*a_buf);



// Sum all the tdblocks into a node then apply sigmoid function

h->sigmoid(*thresh1);


// Step 2): Propagate through to the output nodes

// Buffer the hidden layer activations to
// h_buf[i][tn] for the last K timesteps

buffer( h_buf, h );



//  Sum the K buffered time-delay connections. Each
//  connection is the product of a time-delayed input
//  to the hidden node and the weight matrix. Get
//  tdblocks for each hidden node.


propagate(*o ,*W2,*Tau2,*h_buf);


// Sum all the tdblocks then apply sigmoid function

   if(trace){
     cerr << "\nUnsquashed guess: " << *o
    << "\nOutput layer threshold " << *thresh2;
   }

   o->sigmoid(*thresh2);
```

```
// Step 3): Compute the error for the output layer

  if (epoch){   // adjust weights at the end of the cycle


  // d = o (1-o) (o-t)
  // the somewhat circuitous code is so that we can use existing
  // overloaded operators from the vector class
    *d = (*(v.b) - *o);



    if(trace){
      cerr.precision(6);
      cerr << "\nDesired Output: " << *(v.b) << " Guess: " << *o ;
    }

    maxdiff=d->maxval();
    *d  = *d *  o->d_logistic(); //d_logistic() returns v(1-v)


// Step 4): Compute the error for hidden nodes

    backprop(*e,*d,*W2);

    *e = *e  *  h->d_logistic();  // returns dot product of vec & complement


  // weights will be adjusted at end of cycle with following totals

    *totd += *d;

    *tote += *e;


  }
  else{   // pattern-by-pattern training


// Step 3): Compute the error for the output layer

  // d = o (1-o) (o-t)
  // the somewhat circuitous code is so that we can use existing
```

```
// overloaded operators from the vector class

  *d = (*(v.b) - *o);

MSE += (0.5 * (*d * *d));


  if(trace){
    cerr.precision(6);
    cerr << "\nDesired Output: " << *(v.b) << " Guess: " << *o ;
  }

  maxdiff=d->maxval();
  *d  = *d *  o->d_logistic();  //d_logistic() returns v(1-v)


// Step 4): Compute the error for hidden nodes


   backprop(*e,*d,*W2);
   *e = *e  *  h->d_logistic(); // returns dot product of vec & complement


// Step 5): Update weights and time-delays for the hidden to output layer

  // W2 = W2 + à h d + (momentum*dW2(t-1))
    initdWts(*dW2,*h_buf,*d,learnrate,momentum); // " à h d + momentum" part


    (*W2) += *dW2;


 if(TDNN==0){

     // Tau2 = Tau2 + (rate * h' W2 d)
                deriv(*h_buf_p,*h_buf,*Tau2,tstep);


                initdTau(*dTau2,*W2,*h_buf_p,*d,learnrate2);

                (*Tau2) +=  *dTau2;
 }


    *thresh2 += ( (*d) * learnrate );
```

```
// Step 6): Update weights and time-delays for the input to hidden layer

  // W1 = W1 + à i e + (momentum*dW1(t-1))
    initdWts(*dW1,*a_buf,*e,learnrate,momentum);


        (*W1) += *dW1;


 if(TDNN==0){
        // Tau1 = Tau1 + (rate * i' W1 e)

                deriv(*a_buf_p,*a_buf,*Tau1,tstep);


                initdTau(*dTau1,*W1,*a_buf_p,*e,learnrate2);


                (*Tau1) += *dTau1;

  }

    *thresh1 += ( (*e) * learnrate);


  }       // end pattern-by-pattern training



// Step 7): Compare max difference to tolerance

    if(trace)
    cerr << "\nMaximum difference: " << maxdiff;
    if(maxdiff < tolerance ) { return 1; }
    else {
      return 0;
    }
 } //end encode()


void atnn::deriv(matrix& m2,matrix& m1,mtrx3d& m3d1,const float tstep)
//Compute the derivative of node input for time-delay learning rule updates
{
```

```
 int j=0;  // time-delay value is the same for all output nodes
            // associated with each input
            for(int i=0;i<m3d1.width();i++)
              for(int k=0;k< K;k++){
          int tau = int (m3d1.getval(j,i,k) + 0.5);
          if(tau==0)
           m2.setval(i,k,(((m1.getval(i,tau)) - (m1.getval(i,tau+1)))/tstep));
          else
           m2.setval(i,k,(((m1.getval(i,tau-1)) - (m1.getval(i,tau+1)))/(2*tstep)));
          }

}


void atnn::buffer( matrix *m1, vec *v1 )
//
{
 for(int i=0; i<m1->depth(); i++ )
 {
   for(int tn=K-1;tn>=0;tn--)
   {
     m1->setval(i,tn+1,(m1->getval(i,tn)));
   }
   m1->setval(i,0,(v1->v[i]));
 }

}


vec  atnn::propagate(vec& v1, mtrx3d& m3d1,mtrx3d& m3d2,matrix& m1)
// Double sum of product (weights * node inputs) for each time-delay on each input
{
  for(int j=0;j<m3d1.depth();j++)
            for(int i=0;i<m3d1.width();i++)
                for(int k=0;k<K;k++)
        {
          if ( m3d2.getval(j,i,k) < 0.0 ) {m3d2.setval(j,i,k,0.0);}
          int tau = int (m3d2.getval(j,i,k) + 0.5);
          v1.v[j] += (m3d1.getval(j,i,k)) * (m1.getval(i,tau));
        }
  return v1;
}


vec  atnn::backprop(vec& v1,vec& v2,mtrx3d& m3d1)
```

```
// Double sum for computing back-propagation error to the hidden nodes
{
 for(int j=0;j<m3d1.depth();j++)

        for(int m=0;m<m3d1.width();m++)

              for(int k=0;k<K;k++)

                    v1.v[m] += (v2.v[j]) * (m3d1.getval(j,m,k));

 return v1;

}


void atnn::initdWts(mtrx3d& m3d1,matrix& m1,const vec& v1,
        const float rate,const float momentum)
// Used to initialize a 3d matrix to the element by element product
// of v1 and m1 times the learn rate
// also adding in the previous contents of the 3d matrix
// multiplied by a momentum term.
{
 for(int i=0;i<m3d1.depth();i++)
   for(int j=0;j<m3d1.width();j++)
     for(int k=0;k<K;k++)
        m3d1.setval(i,j,k,(((m3d1.getval(i,j,k))*momentum)+
        ((v1.v[i])*(m1.getval(j,k))*rate)));
}


void atnn::initdTau(mtrx3d& m3d1,mtrx3d& m3d2,matrix& m1,const vec& v1,
          const float rate)
// Used to initialize a 3d matrix to the element by element product
// of v1, m3d2, m1, and the learn rate.  No momentum term.

{
 for(int i=0;i<m3d2.depth();i++)
   for(int j=0;j<m3d2.width();j++)
     for(int k=0;k<K;k++)
        m3d1.setval(i,j,k,((-1)*(v1.v[i])*(m3d2.getval(i,j,k))
              *(m1.getval(j,k))*rate));
}
```

```
vec atnn::recall( vec &v )
{

// Step 1): h = F(W1 i)

  // Buffer inputs to a_buf[i][tn] for the last K timesteps


  buffer( a_buf, &v );


  // Sum the K buffered time-delay connections.  Each connection
  // is the product of a time delayed input and the weight
  // matrix.  Get tdblocks for each input.

  propagate(*h,*W1,*Tau1,*a_buf);


  h->sigmoid(*thresh1);


// Step 2): o = F (W2 h)

  vec out(this->p);

  // Buffer the hidden layer activations to
  // h_buf[i][tn] for the last K timesteps

  buffer( h_buf, h );


  // Sum the K buffered time-delay connections.  Each
  // connection is the product of a time-delayed input
  // to the hidden node and the weight matrix.  Get
  // tdblocks for each hidden node.
  //

     propagate(out,*W2,*Tau2,*h_buf);


     if(trace){
       cerr << "\nUnsquashed guess: " << out
```

```
        << "\nOutput layer threshold " << *thresh2;
    }

    out.sigmoid(*thresh2);

    return out;
}

//
// This will get called from the neural network train since train
// will call the most derived cycle method.
// We need to override the network cycle since the time delay algorithm
// may require the weights to be update at the end of a cycle.

float atnn::cycle(istream& s)
{
  vecpair v(n,p);
  int good = 0;
  int total = 0;


  s >> *minvecs;

  s >> *maxvecs;

  if(epoch)  // initialize error accumulation vectors
  {
        for(int i=0;i<totd->length();i++)
        totd->set(i);
        for(i=0;i<tote->length();i++)
                tote->set(i);
  }

  skipcmt(s);


  int okay = TRUE;
  int td = total;


        while ( td < K )  // Buffer inputs to a_buf[i][tn] for K timesteps
        {
        s >> v;
```

```
if(s.eof()||s.fail())
        okay = FALSE;

if ( okay )
{


        v.scale(*minvecs,*maxvecs);

        buffer( a_buf, v.a ); td++;
}
}


while ( okay )
{
s >> v;


if(s.eof()||s.fail())
        okay = FALSE;

if ( okay )
{


        v.scale(*minvecs,*maxvecs);


        if(encode(v))
        {
                good++;
                if(!trace)
                        cerr << '.';
                else
                        cerr << "\nGood guess";
        }
        else
        {
                if(!trace)
                        cerr <<  'x';
                else
                        cerr << "\nBad guess";
        }
```

```
                total++;
        }

#ifdef __TURBOC__
  if(kbhit()){return 1.0;}
#else
  if( getbutton( ESCKEY ) )
  {
        return 1.0;
  }
#endif

  }

  if(epoch){ // adjust weights at end of cycle

    // W2 = W2 + à h d (total)
    initdWts(*dW2,*h_buf,*totd,learnrate,momentum); // " à h d " part
    (*W2) += *dW2;

    if(TDNN==0){

    //Tau2 = Tau2 + (rate * h' W2 d(total))

      deriv(*h_buf_p,*h_buf,*Tau2,tstep);
      initdTau(*dTau2,*W2,*h_buf_p,*totd,learnrate);
      (*Tau2) += *dTau2;

    }

    *thresh2 += ( (*totd) * learnrate );

    // W1 = W1 + à i e (total)

    initdWts(*dW1,*a_buf,*tote,learnrate,momentum);
    (*W1) += *dW1;


    if(TDNN==0){

      //Tau1 = Tau1 + (rate * i' W1 e(total))

      deriv(*a_buf_p,*a_buf,*Tau1,tstep);
      initdTau(*dTau1,*W1,*a_buf_p,*tote,learnrate);
      (*Tau1) += *dTau1;
```

```
    }

    *thresh1 += ( (*tote) * learnrate );

  }


  avg_MSE  = MSE / float(total);
  accuracy = float(good)/float(total);

  char errfn[32];

  sprintf(errfn,"%s.ERR",name);
  ofstream errf(errfn,ios::app);

  errf << cycleno << "\t" << avg_MSE << "\t" << accuracy << endl;
  errf.close();


  MSE=0.0;

  cerr << "\n" << "avg_MSE is " << avg_MSE;

  cerr << "\n" << accuracy * 100 << " percent correct.\n";
  return accuracy;
}


//////////////////////////////////////////////////////////
//
// ADAPTIVE TIME DELAY ALGORITHM METHODS - TEST AND RUN
//

float atnn::test()
{
  int good = 0;
  int total = 0;
  char tstfn[32];
  vecpair v(n,p);
  vec out(p);

  if(!loadweights())
  {
    cout << "No stored network to test.";
```

```
  return 0;
}

sprintf(tstfn,"%s.TST",name);
ifstream tstf(tstfn,ios::in);


tstf >> *minvecs;

tstf >> *maxvecs;


// skip comment
skipcmt(tstf);


int okay = TRUE;
int td = total;


        while ( td < K ) // Buffer inputs to a_buf[i][tn] for K timesteps
        {
        tstf >> v;


        if(tstf.eof()||tstf.fail())
                okay = FALSE;

        if ( okay )
        {
                v.scale(*minvecs,*maxvecs);
                buffer( a_buf, v.a ); td++;
        }
        }

        while ( okay )
        {
        tstf >> v;

        if(tstf.eof()||tstf.fail())
                okay = FALSE;

        if ( okay )
        {
```

```
                v.scale(*minvecs,*maxvecs);
                        out=recall(*(v.a));
                        if( (*(v.b)-out).maxval() < tolerance )
                        good++;

                tMSE = (0.5 * ((*(v.b)-out) * (*(v.b)-out)));

                MSE += tMSE;

                        total++;
                }

                char tsterrfn[32];

                sprintf(tsterrfn,"%s.TER",name);
                ofstream tsterrf(tsterrfn,ios::app);

                tsterrf << total << "\t" << *(v.b) << "\t" << out
                        << "\t" <<  tMSE << endl;
                tMSE = 0.0;

                tsterrf.close();



        }

        char tsterrfn[32];

        sprintf(tsterrfn,"%s.TER",name);
        ofstream tsterrf(tsterrfn,ios::app);

        tsterrf << "EPOCH # : " << cycleno << "\t"
          << "time avg MSE = " << MSE/total << endl;
        MSE = 0.0;
        tsterrf.close();

        cerr << "\n" << float(good)/float(total) * 100 << " percent correct.\n";
        return float(good)/float(total);

}



void atnn::run()
```

```
{
  char ifn[16],ofn[16];

  vec in(n),out(p),minvec(n),maxvec(n);

  if(!loadweights()){
    cout << "No stored network to run.\n";
    return;
  }

  sprintf(ifn,"%s.IN",name);
  sprintf(ofn,"%s.OUT",name);
  cout << "Running from " << ifn << "\n";
  cout << "Output to " << ofn << "\n";
  ifstream inf(ifn,ios::in);
  ofstream outf(ofn,ios::out);

  inf >> minvec;
  inf >> maxvec;

  // skip comment
  skipcmt(inf);

  int okay = TRUE;
  int td = 0;


        while ( td < K ) // Buffer inputs to a_buf[i][tn] for K timesteps
        {
        inf >> in;

        if(inf.eof()||inf.fail())
                okay = FALSE;

        if ( okay )
        {
                in.scale(minvec,maxvec);
                buffer( a_buf, &in ); td++;
        }
        }

        while ( okay )
        {
        inf >> in;
```

```
        if(inf.eof()||inf.fail())
                okay = FALSE;

        if ( okay )
        {
                in.scale(minvec,maxvec);
                        outf << recall(in);
        }
  }
  return;
}



////////////////////////////////////////
//
// ATNN LEVEL INPUT/OUTPUT METHODS:
// Saving and loading weights, skipping comments.
//

int atnn::saveweights()
{
  FILE *f;
  char fn[32];

  sprintf(fn,"%s.WTS",name);
  f=fopen(fn,"wb");

#ifdef __TURBOC__
  if(f <= 0)  // couldn't open the file
  {
#else
  if(f == NULL)  // couldn't open the file
  {
#endif
  cerr << "Open of file " << fn << " save failed.\n";
  return 0;
  }
  else
  {

  }

  fwrite(&cycleno,sizeof(int),1,f);
```

```
  if( !(W1->save(f))
    || !(W2->save(f))
    || !(Tau1->save(f))
    || !(Tau2->save(f))
    || !(thresh1->save(f))
    || !(thresh2->save(f))
   )
  {
  cerr << "Nothing to save in file " << fn << ".\n";
  fclose(f);
  return 0;
  }
  else
  {
  cerr << "Saved Weights and Taus in file " << fn << ".\n";
  fclose(f);
  }

  // put matrices into ".MAT" in readable form

  if(trace){
    sprintf(fn,"%s.MAT",name);
    ofstream matf(fn,ios::out);
    matf << "First weight matrix contains: \n"
        << *W1
        << "First time-delay matrix contains: \n"
        << *Tau1
    << "Second weight matrix contains: \n"
    << *W2
    << "Second time-delay matrix contains: \n"
    << *Tau2;
  }
  return 1;
}

int atnn::loadweights()
{
  FILE *f;
  char fn[32];

  int ret_val = FALSE;

  sprintf(fn,"%s.WTS",name);
  f=fopen(fn,"rb");
```

```
#ifdef __TURBOC__
  if(f <= 0)  // couldn't open the file
  {
#else
  if(f == NULL)  // couldn't open the file
  {
#endif

  }
  else
  {
  ret_val = TRUE;
  }

  if ( ret_val )
  {
  fread(&cycleno,sizeof(int),1,f);

          if(  !(W1->load(f))
   || !(W2->load(f))
            || !(Tau1->load(f))
            || !(Tau2->load(f))
            || !(thresh1->load(f))
            || !(thresh2->load(f)) )
          {
                  ret_val = FALSE;
          }
  else
  {
                  ret_val = TRUE;
  }
          fclose(f);
  }


  return ret_val;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////
// NET.CC
// Source code for abstract neural network base class

#include "net.hpp"

#ifndef __TURBOC__

#define _MAX_PATH  16

#endif


////////////////////////////////
// Parameter table functions

int readparms(int n,PARM *p,char *name)
{
  char fn[16];
  sprintf(fn,"%s.DEF",name);
  ifstream def(fn,ios::in);
  if(!def){
    cerr << "Failed to find definition file.\n";
    return 0;
  }
  while  (readparm(def,n,p) && !def.eof())
    ;
  return 0;
}

istream& readparm(istream& s,int noparms,PARM *p)
// This streams extraction operator takes input from network definition file
// for one definition parameter. It reads in the name of the parameter
// and then looks up which entry in the parameter table to instantiate
// with a value.
{
  char keyword[NAMELEN],val[16];
  s >> keyword;
  if(!s || s.eof() || s.fail())  // end of file or failure to read keyword
    return s;

  for(int i=0;i<noparms;i++)
    if(!strcmp(keyword,p[i].name))
      break;
```

```
 if(i < noparms) // recognized parameter
   switch(p[i].type){
     case string:  s >> p[i].val.s; break;
     case integer: s >> p[i].val.i; break;
     case real:    s >> p[i].val.f; break;
   }
 else
   s >> val;

 return s;
}



////////////////////////////////////////////////////////////////
//              NET CLASS
// Abstract neural net class methods
//

net::net(char *s)
{
  char fn[16];
  name=new char[strlen(s)+1];
  strcpy(name,s);
  const NOPARMS=5;


  PARM parms[NOPARMS];

  strcpy(parms[0].name, "INPUTS");
  parms[0].type = integer;

  strcpy(parms[1].name, "OUTPUTS");
  parms[1].type = integer;

  strcpy(parms[2].name, "RATE");
  parms[2].type = real;

  strcpy(parms[3].name, "DECAY");
  parms[3].type = real;

  strcpy(parms[4].name, "ITERS");
  parms[4].type = integer;
```

```
  readparms(NOPARMS,parms,name);
  n        = parms[0].val.i;
  p        = parms[1].val.i;
  learnrate = parms[2].val.f;
  decayrate = parms[3].val.f;
  iters    = parms[4].val.i;
  return;
}

net::~net()
{
  delete name;
}

void net::train()
{
  ifstream *s;
  float ret;

  char fn[_MAX_PATH];
  sprintf(fn,"%s.FCT",name);



  if( loadweights() )
  {
    cerr << "Training from stored weights.\n";
  }
  else
  {
    cerr << "Training from new weights.\n";
  }

#ifdef __TURBOC__
  cerr << "Training from " << fn << ". Press any key to stop.\n";
#else
  cerr << "Training from " << fn << ". Press ESC key to stop.\n";
#endif

  int okay = TRUE;
  while( okay )
  {
    s=new ifstream(fn,ios::in);

    if(!*s){
```

```
      cerr << "Failed to open fact file.\n";
      return;
    }

    cerr << "Cycle " << ++cycleno << ": ";
    if ( cycleno >= iters ) { okay = FALSE; }

    ret=cycle(*s);
    delete s;

#ifdef __TURBOC__
    if(ret>=1.0 || kbhit())
    {
#else
    if(ret>=1.0 || getbutton( ESCKEY ) )

    {
#endif
      cerr << "Training suspended at " << cycleno << " cycles.\n";
      okay = FALSE;
    }
  }

  saveweights();
  return;
}



float net::cycle(istream& s)
{
  vecpair v(n,p);
  int good = 0;
  int total = 0;

  skipcmt(s);
  for(;;){
    s >> v;
    if(s.eof()||s.fail())break;
    if(encode(v))
      good++;
    total++;
  }
```

```
  return float(good)/float(total);
}

int net::skipcmt(istream& inf)
{
  int c;
  inf.unsetf(inf.skipws);
  if(inf.peek()==':'){
    do{
      c=inf.get();
      if(c<0)
        return 0;
    } while( (c!=0xd) && (c!=0xa) );
    inf.setf(inf.skipws);
    return 1;
  }
  else{
    inf.setf(inf.skipws);
    return 0;
  }
}


float net::test()
{
  int good = 0;
  int total = 0;
  char tstfn[32];
  vecpair v(n,p);
  vec out(p);

  if(!loadweights())
  {
    cout << "No stored network to test.";
    return 0;
  }

  sprintf(tstfn,"%s.TST",name);
  ifstream tstf(tstfn,ios::in);

  // skip comment
  skipcmt(tstf);


  for(;;){
```

```
    if(!(tstf>>v))break;
    out=recall(*(v.a));
    if( (*(v.b)-out).maxval() < tolerance )
      good++;
    total++;
  }

  cerr << "\n" << float(good)/float(total) * 100 << " percent correct.\n";
  return float(good)/float(total);
}

void net::run()
{
  char ifn[16],ofn[16];
// int c;
  vec in(n),out(p);

  if(!loadweights()){
    cout << "No stored network to run.\n";
    return;
  }

  sprintf(ifn,"%s.IN",name);
  sprintf(ofn,"%s.OUT",name);
  cout << "Running from " << ifn << "\n";
  cout << "Output to " << ofn << "\n";
  ifstream inf(ifn,ios::in);
  ofstream outf(ofn,ios::out);

  skipcmt(inf);
  for(;;){
    if(!(inf>>in))break;;
    if(!inf || inf.eof()|| inf.fail())break;
    outf << recall(in);
  }
  return;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
// VECMAT.CC
// vector and matrix class methods
// Author:  Capt James Gainey, GEO-93D       Last Modified:  10 Sep 93
// Modified from VECMAT.CPP  Adam Blum (1990)


#include <vecmat.hpp>


////////////////////////////////
// vector class member functions

vec::vec(int size,int val)
{
  v = new float[n=size];
      for(int i=0;i<n;i++)
            v[i]=val;
} // constructor

vec::~vec() { delete v;} // destructor
vec::vec(vec& v1) // copy-initializer
{
      v=new float[n=v1.n];
      for(int i=0;i<n;i++)
            v[i]=v1.v[i];
}

vec& vec::operator=(const vec& v1)
{
      delete v;
      v=new float[n=v1.n];
      for(int i=0;i<n;i++)
            v[i]=v1.v[i];
      return *this;
}

vec vec::operator+(const vec& v1)
{
      vec sum(v1.n);
      for(int i=0;i<v1.n;i++)
            sum.v[i]=v1.v[i]+v[i];
      return sum;
}
```

```
vec vec::operator+(const float d)
{
      vec sum(n);
      for(int i=0;i<n;i++)
            sum.v[i]=v[i]+d;
      return sum;
}



vec& vec::operator+=(const vec& v1)
{
  for(int i=0;i<v1.n;i++)
            v[i]+=v1.v[i];
      return *this;
}

float vec::operator*(const vec& v1) // dot-product
{
      float sum=0;
      for(int i=0;i<min(n,v1.n);i++)
            sum+=(v1.v[i]*v[i]);
      return sum;
}

int vec::operator==(const vec& v1)
{
      if(v1.n!=n)return 0;
      for(int i=0;i<min(n,v1.n);i++){
            if(v1.v[i]!=v[i]){
                  return 0;
            }
      }
      return 1;
}

float vec::operator[](int x)
{
      if(x<length() && x>=0)
            return v[x];
  else
            cerr << "vec index out of range";
      return 0;
}
```

```
int vec::length(){return n;} // length method

vec& vec::garble(float noise) // corrupt vector w/random noise
{
      time_t t;
      time(&t);
      srand((unsigned)t);
      for(int i=0;i<n;i++){
            if((rand()%10)/10<noise)
       v[i]=1-v[i];
      }
      return *this;
}


vec& vec::normalize() // normalize by length
{
      for(int i=0;i<n;i++)
            v[i]/=n;
      return *this;
}
vec& vec::normalizeon() //normalize by nonzero elements
{
  int on=0;
      for(int i=0;i<n;i++)
            if(v[i])
                  on++;
      for(i=0;i<n;i++)
            v[i]/=on;
      return *this;
}

vec& vec::randomize(float range)
{
      time_t t;
      int pct,val,rnd;
      if(range){
     time(&t);
            srand((unsigned)t);
      }
      for(int i=0;i<n;i++){
            rnd=rand();
            pct=(int) (range * 100.0);
            val= rnd % pct;
            v[i]= (float) val / 100.0 ;
```

```
            if(range<0)
                v[i] = fabs(range) - (v[i] * 2.0);
        }
        return *this;
}

float vec::maxval() // returns maximum ABSOLUTE value
{
        float mx=0;
        for(int i=0;i<n;i++)
            if(fabs(v[i])>mx){
                mx=fabs(v[i]);
            }
        return mx;
}

vec& vec::scale(vec& minvec,vec& maxvec)
{
        for(int i=0;i<n;i++){
            if(v[i]<minvec.v[i])
                v[i]=0;
            else if(v[i]>maxvec.v[i])
                v[i]=1;
            else if((maxvec.v[i]-minvec.v[i])==0)
                v[i]=1;
            else
                v[i]=(v[i]-minvec.v[i])/(maxvec.v[i]-minvec.v[i]);
        }
        return *this;
}

float vec::d_logistic() // returns vec * (1-vec)
{
        float sum=0.0;
    for(int i=0;i<n;i++)
            sum+=(v[i]*(1-v[i]));
        return sum;
}

// Euclidean distance function ||A-B||
float vec::distance(vec& A)
{
        float sum=0,d;
        for(int i=0;i<n;i++){
            d=v[i]-A.v[i];
```

```
            if(d)sum+=pow(d,2);
        }
        return sum?pow(sum,0.5):0;
}

// index of the highest item in vector
int vec::maxindex()
{
        int idx,i;
        float mx;
        for(i=0,mx=-INT_MAX;i<n;i++)
                if(v[i]>mx){
                        mx=v[i];
                        idx=i;
                }
        return idx;
}



double logistic(double activation)
{
        /* These underflow limits were copied from McClelland's bp implementation.
                We had problems with underflow with numbers that should have been
                small enough in magnitude. McClelland seems to have encountered this
                and established the numbers below as reasonable limits. - AB */
        if(activation>11.5129)
                return 0.99999;
        if(activation<-11.5129)
                return 0.00001;
        return 1.0/(1.0+exp(-activation));
}

vec& vec::getstr(char *s)
{
        for(int i=0;i<MAXVEC&&s[i];i++){
                if(isalpha(s[i]))
                        v[toupper(s[i])-'A']=1;
        }
        return *this;
}

void vec::putstr(char *s)
{
        int ct=0;
```

```
        for(int i=0;i<26;i++)
            if(v[i]>0.9)
                s[ct++]='A'+i;
}

vec vec::operator-(const vec& v1)
{
        vec diff(n);
        for(int i=0;i<n;i++)
            diff.v[i]=v[i]-v1.v[i];
        return diff;
}

vec vec::operator-(const float d)       // subtraction of constant
{
        vec diff(n);
        for(int i=0;i<n;i++)
            diff.v[i]=v[i]-d;
        return diff;
}

vec vec::operator*(float c)
{
        vec prod(length());
        for(int i=0;i<prod.n;i++)
            prod.v[i]=v[i]*c;
        return prod;
}

vec& vec::operator*=(float c)
{
        for(int i=0;i<n;i++)
            v[i]*=c;
        return *this;
} // vector multiply by constant


const SCALE=1;

vec& vec::sigmoid(vec& thresh)
// this is the sigmoid activation function we have chosen for
// our backprop implementation. It happens to use the logistic
// function: 1/(1+e^-x)
{
        for(int i=0;i<n;i++)
```

```
            v[i] = (float) logistic( (double) (SCALE * (v[i]+thresh[i])) );
        return *this;
}

vec& vec::set(int i,float f)
{
        v[i]=f;
        return *this;
}

istream& operator>>(istream& s,vec& v1)
// format: list of floating point numbers followed by ','
{
        float d;int i=0,c;
        for(;;){

                s>>d;
                if(s.eof())
                        return s;
                if(s.fail()){
                        s.clear();
                        do
                                c=s.get();
                        while(c!=',' && c);
                        return s;
                }
                v1.v[i++]=d;
                if(i==v1.n){
                        do
                                c=s.get();
                        while(c!=',');
                        return s;
                }
        }
}


ostream& operator<<(ostream& s,vec& v1)
// format: list of floating point numbers followed by ','
{
        s.precision(6);
        for(int i=0;i<v1.n;i++)
                s << v1[i] <<" ";
        s << ",";
        return s;
```

```
}


int vec::save(FILE *f) // save binary values of matrix to specified file
{
  int success=1;
  for(int i=0;i<n;i++)
      if(fwrite(&(v[i]),sizeof(v[i]),1,f) < 1)
        success=0;
  return success;
}

int vec::load(FILE *f) // load binary values of matrix from specified file
{
  int success=1;
  for(int i=0;i<n;i++)
      if(fread(&(v[i]),sizeof(v[0]),1,f) < 1)
        success=0;
  return success;
}



////////////////////////////
// matrix  member functions
matrix::matrix(int n,int z,float range)
{
      int i,j,rnd;time_t t;
      int pct;
      m=new float *[n];
      if(range){
            time(&t);
            srand((unsigned)t);
      }
      for(i=0;i<n;i++){
            m[i]=new float[z];
            for(j=0;j<z;j++){
                  if(range){
                        rnd=rand();
                           pct=(int) (range * 100.0);
                        m[i][j]= (float)(rnd % pct) / 100.0 ;
                        if(range<0)
                              m[i][j] = fabs(range) - (m[i][j] * 2.0);
                  }
                  else
```

```
                    m[i][j]=0;
            }
        }
        r=n;
  c=z;
}

matrix::matrix(int n,int p,float value,float range)
{
        int i,j;
        i=int(range);
        m=new float *[n];
        for(i=0;i<n;i++){
                m[i]=new float[p];
                for(j=0;j<p;j++)
                        m[i][j]=value;
        }
        r=n;
  c=p;
}

matrix::matrix(int n,int p,char *fn)
{
        int i;
//int j,rnd;
//time_t t;
        m=new float *[n];
        for(i=0;i<n;i++){
        m[i]=new float[p];
        }
        r=n;
        c=p;
        ifstream in(fn,ios::in);
        in >> *this;
}

matrix::matrix(const vecpair& vp)
{
        int j;
        r=vp.a->length();
        c=vp.b->length();
        m=new float *[r];
        for(int i=0;i<r;i++){
                m[i]=new float[c];
                for(j=0;j<c;j++)
```

```
                m[i][j]=((vp.a)->v[i])*((vp.b)->v[j]);
        }
}// constructor

matrix::matrix(vec& v1,vec& v2)
{
        int j;
        r=v1.length();
        c=v2.length();
        m=new float *[r];
        for(int i=0;i<r;i++){
                m[i]=new float[c];
                for(j=0;j<c;j++)
                        m[i][j]=v1.v[i]*v2.v[j];
        }
}// constructor

matrix::matrix(matrix& m1) // copy-initializer
{
        r=m1.r;
        c=m1.c;
        m=new float *[r];
        for(int i=0;i<r;i++){
                m[i]=new float[c];
                for(int j=0;j<c;j++)
                        m[i][j]=m1.m[i][j];
        }
}

matrix::~matrix()
{
        delete []m;
} // destructor

matrix& matrix::operator=(const vecpair& vp)
{
        int j;double d;
        r=vp.a->length();
        c=vp.b->length();
        for(int i=0;i<r;i++){
                for(j=0;j<c;j++){
                        d=((vp.a)->v[i])*((vp.b)->v[j]);
                        m[i][j]=(float)d;
                }
        }
```

```
    return *this;
}

matrix& matrix::operator=(const matrix& m1)
{
    for(int i=0;i<r;i++)
        delete m[i];
    r=m1.r;
    c=m1.c;
    m=new float*[r];
    for(i=0;i<r;i++){
        m[i]=new float[c];
        for(int j=0;j<r;j++)
            m[i][j]=m1.m[i][j];
    }
    return *this;
}

matrix matrix::operator+(const matrix& m1)
{
    int i,j;
    matrix sum(r,c);
    for(i=0;i<r;i++)
        for(j=0;j<r;j++)
            sum.m[i][j]=m1.m[i][j]+m[i][j];
    return sum;
}


matrix& matrix::operator*(const float d)
{
    int i,j;
    for(i=0;i<r;i++)
        for(j=0;j<c;j++)
            m[i][j]*=d;
    return *this;
}

vec matrix::colslice(int col)
{
    vec temp(r);
    for(int i=0;i<r;i++)
        temp.v[i]=m[i][col];
    return temp;
}
```

```
vec matrix::rowslice(int row)
{
      vec temp(c);
      for(int i=0;i<c;i++)
            temp.v[i]=m[row][i];
      return temp;
}

void matrix::insertcol(vec& v,int col)
{
      for(int i=0;i<v.n;i++)
            m[i][col]=v.v[i];
}

void matrix::insertrow(vec& v,int row)
{
      for(int i=0;i<v.n;i++)
            m[row][i]=v.v[i];
}

int matrix::depth(){return r;}
int matrix::width(){return c;}

float matrix::getval(int row,int col)
{
  return m[row][col];
}

void matrix::setval(int row,int col,float val)
{
  m[row][col] = val;
}

int matrix::closestcol(vec& v)
{
      int mincol;
      float d;
      float mindist=INT_MAX;
      vec w(r);
      for(int i=0;i<c;i++){
            w=colslice(i);
            if( (d=v.distance(w)) < mindist){
                  mindist=d;
                  mincol=i;
            }
```

```
        }
        return mincol;
}
int matrix::closestrow(vec& v)
{
        int minrow;
        float d;
        float mindist=INT_MAX;
        vec w(c);
        for(int i=0;i<r;i++){
                w=rowslice(i);
                if( (d=v.distance(w)) < mindist){
                        mindist=d;
                        minrow=i;
                }
        }
        return minrow;
}
int matrix::closestrow(vec& v,int *wins,float scaling)
{
        int minrow;
        float d;
        float mindist=INT_MAX;
        vec w(c);
        for(int i=0;i<r;i++){
                w=rowslice(i);
                d=v.distance(w);
                d*=(1+((float)wins[i]*scaling));
                if( d < mindist){
                        mindist=d;
                        minrow=i;
                }
        }
        return minrow;
}

int matrix::save(FILE *f) // save binary values of matrix to specified file
{
  int success=1;
  for(int i=0;i<r;i++)
    for(int j=0;j<c;j++)
      if(fwrite(&(m[i][j]),sizeof(m[0][0]),1,f) < 1)
                        success=0;
  return success;
}
```

```
int matrix::load(FILE *f) // load binary values of matrix from specified file
{
      int success=1;
      for(int i=0;i<r;i++)
            for(int j=0;j<c;j++)
                  if(fread(&(m[i][j]),sizeof(m[0][0]),1,f) < 1)
                        success=0;
      return success;
}

#ifdef __TURBOC__
int _Cdecl intcmp(const void* i1,const void *i2)
#else
int intcmp(const void* i1,const void *i2)
#endif
{
      if(*(int *)i1 > *(int *)i2)
         return 1;
      if(*(int *)i1 < *(int *)i2)
         return -1;
      return 0;
}


matrix& matrix::operator+=(const matrix& m1)
{
      int i,j;
      for(i=0;i<r&&i<m1.r;i++)
            for(j=0;j<c&&j<m1.c;j++)
                  m[i][j]+=(m1.m[i][j]);
      return *this;
}

matrix& matrix::operator*=(const float d)
{
      int i,j;
      for(i=0;i<r;i++)
            for(j=0;j<c;j++)
                  m[i][j]*=d;
      return *this;
}

vec matrix::operator*(vec& v1)
{
```

```
        vec temp(v1.n==r?c:r),temp2(v1.n==r?r:c);
        for(int i=0;i<((v1.n==r)?c:r);i++){
            if(v1.n==r)
                temp2=colslice(i);
            else
                temp2=rowslice(i);
            temp.v[i]=v1*temp2;
        }
        return temp;
    }


    void matrix::initvals(const vec& v1,const vec& v2,const float rate, const float
momentum)
    {
      int j;
      for(int i=0;i<r;i++)
        for(j=0;j<c;j++)
          m[i][j]=(m[i][j]*momentum)+((v1.v[i]*v2.v[j])*rate);
    }

    ostream& operator<<(ostream& s,matrix& m1)
    // print a matrix
    {
        for(int i=0;i<m1.r;i++){
            for(int j=0;j<m1.c;j++){
                s << m1.m[i][j] << " ";
            }
            s << "\n";
        }
        return s;
    }

    istream& operator>>(istream& s,matrix& m1)
    {
        for(int i=0;i<m1.r;i++){
            for(int j=0;j<m1.c;j++){
                s >> m1.m[i][j];
            }
        }
        return s;

    }

    ////////////////////////////////////////
```

```
// vecpair  member functions

//////////////////
// constructors




vecpair::vecpair(int n,int p,int val)
{
      a=new vec(n,val);b=new vec(p,val);
}

vecpair::vecpair(vec& A,vec& B)
{
      a=new vec(A.length());
      *a=A;
      b=new vec(B.length());
      *b=B;
}

vecpair::vecpair(const vecpair& AB) // copy-initializer
{
      a=new vec((AB.a)->length());
      b=new vec((AB.b)->length());
      *a=*(AB.a);
      *b=*(AB.b);
}

vecpair::~vecpair() {
      delete a; delete b;
} // destructor

vecpair& vecpair::operator=(const vecpair& v1)
{
      *a=*(v1.a);
      *b=*(v1.b);
      return *this;
}

vecpair& vecpair::scale(vecpair& minvecs,vecpair& maxvecs)
{
      a->scale(*(minvecs.a),*(maxvecs.a));
```

```
        b->scale(*(minvecs.b),*(maxvecs.b));
        return *this;
}

int vecpair::operator==(const vecpair& v1)
{
        return  (*a == *(v1.a)) && (*b == *(v1.b));
}



istream& operator>>(istream& s,vecpair& v1)
// input a vector pair
{
        s>>*(v1.a)>>*(v1.b);
        return s;
}

ostream& operator<<(ostream& s,vecpair &v1)
// print a vector pair
{
        return s<<*(v1.a)<<*(v1.b)<<"\n";
}


////////////////////////////////
// 3d matrix  member functions

// constructors

mtrx3d :: mtrx3d(int n,int p,int m,float range)
{
  int i,j,k,rnd;time_t t;
        int pct;
        m3d=new float **[n];
        if(range){
                time(&t);
                srand((unsigned)t);
        }
        for(i=0;i<n;i++){
     m3d[i]=new float*[p];
                for(j=0;j<p;j++){
                 m3d[i][j]=new float[m];
                                for(k=0;k<m;k++){
                                if(range){
```

```
                rnd=rand();
                pct=(int) (range * 100.0);
                m3d[i][j][k]= (float)(rnd % pct) / 100.0 ;
                if(range<0)
                  m3d[i][j][k] = fabs(range) - (m3d[i][j][k] * 2.0);
              }
              else
                m3d[i][j][k]=0;
            }
          }
      }
      r=n;
  c=p;
      z=m;
}

mtrx3d::mtrx3d(int n,int p,int m,int irange)
{ // constructor
  int value,i,j,k;
  m3d=new float **[n];
      for(i=0;i<n;i++){
        m3d[i]=new float*[p];
        for(j=0;j<p;j++){
      m3d[i][j]=new float[m];
          value=0;
          for(k=0;k<m;k++){
            if(irange==m){
            m3d[i][j][k]=value;
            value++;
            }
           else
            m3d[i][j][k]=0;
          }
        }
      }

  r=n;
  c=p;
      z=m;
}


mtrx3d::mtrx3d(int n,int p,int m,float value,float range)
{ // constructor
      int i,j,k;
```

```
        i=int(range);
        m3d=new float **[n];
        for(i=0;i<n;i++){
          m3d[i]=new float*[p];
          for(j=0;j<p;j++){
        m3d[i][j]=new float[m];
            for(k=0;k<m;k++)
             m3d[i][j][k]=value;
          }
        }
  r=n;
  c=p;
        z=m;
}

mtrx3d::mtrx3d(int n,int p,int m,char *fn)
{ // constructor
        int i, j;
//int k,rnd;
//time_t t;
        m3d=new float **[n];
        for(i=0;i<n;i++){
     m3d[i]=new float*[p];
     for(j=0;j<m;j++){
            m3d[i][j]=new float[m];
          }
        }
        r=n;
  c=p;
  z=m;
        ifstream in(fn,ios::in);
        in >> *this;
}

mtrx3d::mtrx3d(mtrx3d& m3d1) // copy-initializer
{
        r=m3d1.r;
   c=m3d1.c;
        z=m3d1.z;
        m3d=new float **[r];
        for(int i=0;i<r;i++){
     m3d[i]=new float*[c];
     for(int j=0;j<c;j++){
        m3d[i][j]=new float[z];
        for(int k=0;k<z;k++)
```

```
        m3d[i][j][k]=m3d1.m3d[i][j][k];
          }
        }
}

mtrx3d::~mtrx3d()
{

  delete []m3d;
} // destructor

int mtrx3d::depth(){return r;}
int mtrx3d::width(){return c;}
int mtrx3d::height(){return z;}

mtrx3d& mtrx3d::operator=(const mtrx3d& m3d1)
{
  for(int i=0;i<r;i++)
    for(int j=0;j<r;j++)
      delete m3d[i][j];
      r=m3d1.r;
  c=m3d1.c;
      z=m3d1.z;
      m3d=new float**[r];
      for(i=0;i<r;i++){
        m3d[i]=new float*[c];
        for(int j=0;j<r;j++){
      m3d[i][j]=new float[z];
          for(int k=0;k<r;k++)
        m3d[i][j][k]=m3d1.m3d[i][j][k];
        }
      }
      return *this;
}

mtrx3d mtrx3d::operator+(const mtrx3d& m3d1)
{
      int i,j,k;
  mtrx3d sum(r,c,z);
      for(i=0;i<r;i++)
    for(j=0;j<r;j++)
      for(k=0;k<r;k++)
          sum.m3d[i][j][k]=m3d1.m3d[i][j][k]+m3d[i][j][k];
      return sum;
}
```

```
int mtrx3d::save(FILE *f) // save binary values of 3d matrix to specified file
{
  int success=1;
  for(int i=0;i<r;i++)
    for(int j=0;j<c;j++)
  for(int k=0;k<z;k++)
    if(fwrite(&(m3d[i][j][k]),sizeof(m3d[0][0][0]),1,f) < 1)
                        success=0;
  return success;
}

int mtrx3d::load(FILE *f) // load binary values of 3d matrix from specified file
{
      int success=1;
      for(int i=0;i<r;i++)
    for(int j=0;j<c;j++)
      for(int k=0;k<z;k++)
        if(fread(&(m3d[i][j][k]),sizeof(m3d[0][0][0]),1,f) < 1)
                        success=0;
      return success;
}


float mtrx3d::getval(int row,int col,int z)
{
  return m3d[row][col][z];
}

void mtrx3d::setval(int row,int col,int z,float val)
{
  m3d[row][col][z] = val;
}

mtrx3d& mtrx3d::operator+=(const mtrx3d& m3d1)
{
      int i,j,k;
      for(i=0;i<r&&i<m3d1.r;i++)
    for(j=0;j<c&&j<m3d1.c;j++)
      for(k=0;k<z&&k<m3d1.z;k++)
        m3d[i][j][k]+=(m3d1.m3d[i][j][k]);
      return *this;
```

```
}

mtrx3d& mtrx3d::operator*(const float d)
{
      int i,j,k;
      for(i=0;i<r;i++)
    for(j=0;j<c;j++)
       for(k=0;k<z;k++)
             m3d[i][j][k]*=d;
      return *this;
}


mtrx3d& mtrx3d::operator*=(const float d)
{
      int i,j,k;
      for(i=0;i<r;i++)
    for(j=0;j<c;j++)
      for(k=0;k<z;k++)
            m3d[i][j][k]*=d;
      return *this;
}




ostream& operator<<(ostream& s,mtrx3d& m3d1)
// print a 3d matrix
{
      for(int i=0;i<m3d1.r;i++){
        for(int j=0;j<m3d1.c;j++){
       for(int k=0;k<m3d1.z;k++){
           s << m3d1.m3d[i][j][k] << " ";
           }
        }
            s << "\n";
      }
      return s;
}

istream& operator>>(istream& s,mtrx3d& m3d1)
{
      for(int i=0;i<m3d1.r;i++)
    for(int j=0;j<m3d1.c;j++)
      for(int k=0;k<m3d1.c;k++)
```

```
            s >> m3d1.m3d[i][j][k];

        return s;

}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//ATNN.H
// Header file for Adaptive Time Delay Neural Net implementation
// Developed with Turbo C++ 3.0
// Author:  Capt James Gainey, GEO-93D       Last Modified:  10 Sep 93



#include"net.h"


class atnn: public net {  // adaptive time delay network derived from
private:
  int q;  // size of hidden layer
      int K;  // max # of time delays (range of taus)
      mtrx3d *W1,*W2;  // synapse weight matrices
  mtrx3d *dW1,*dW2;  // used to compute changes to matrices
  mtrx3d *Tau1,*Tau2;  // synapse time delay matrices
  mtrx3d *dTau1,*dTau2;  // used to compute changes to time delay matrices

          matrix *a_buf,*h_buf;  //buffers input to nodes over all time-delays
  matrix *a_buf_p,*h_buf_p;  //buffer for derivative of inputs to
                             //nodes used to compute delta tau
      vec *h,*o,*d,*e,*thresh1,*thresh2,*in;


      int epoch, TDNN;
      vec *totd,*tote;
      vecpair *minvecs,*maxvecs;
  float tstep,momentum,initrange,learnrate2,MSE,avg_MSE,accuracy,tMSE;

      // private member functions
      // these are helper member functions
  void buffer(matrix *m1,vec *v1);
  void deriv(matrix& m2,matrix& m1,mtrx3d& m3d1,
                  const float tstep=1.0f);
  vec  propagate(vec& v1,mtrx3d& m3d1,mtrx3d& m3d2,matrix& m1);
  vec  backprop(vec& v1,vec& v2,mtrx3d& m3d1);

  void initdWts(mtrx3d& m3d,matrix& m1,const vec& v1,
            const float rate=1.0,const float momentum=0.0);
  void initdTau(mtrx3d& m3d1,mtrx3d& m3d2,matrix& m1,const vec& v1,
            const float rate=1.0);

  int saveweights();
```

```
  int loadweights();
  float cycle(istream& s);



public:
      // public member functions
  atnn(char *s);          // constructs based on <name>.DEF file
  ~atnn();               // destructor
      // override pure virtual functions
      int encode(vecpair& v); // store one pattern pair
      vec recall(vec &v);     // recall an output pattern given an input
  float test();
  void run();
};
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
// NET.H
// Header file for abstract neural network base class

// To be used as parent to specific neural network implementations.
// The encode and recall methods are defined as pure virtual functions
// making this an abstract class than can never be instantiated.
// Details of encode and recall must depend on the topology
// itself. However the methods "train", "test", and "run"
// can be defined since they are substantively the same for each
// of the classes. The constructor can be defined and will be used
// by child classes in their own constructors to instantiate
// common elements of derived classes.

#include "vecmat.h"

// parameter class used to point to variable to be initialized
// and specify string to be used in definition file to initialize it

enum vartype {real,integer,string};
const NAMELEN=16;

typedef struct {
  char name[16]; /* string to init value */
  vartype type;
  union {
    char s[8];
    float f;
    int i;
  } val;
} PARM;

istream& readparm(istream& s,int noparms,PARM *p);
int readparms(int n,PARM *p,char *name);

////////////////////////////////////////////////////
//          NET CLASS
//
class net {
protected:
      char *name;  // string used as basename for files
      int n; // size of input layer
      int p; // size of output layer
      float learnrate; // learning rate (defined as 1 where not gradual)
      float decayrate;       // decay (default constructed zero if not applicable)
```

```
        float tolerance;
        int iters;
        int cycleno;
    vecpair *minvecs,*maxvecs;

        // weight saving methods since we don't know topology
        // they must be pure virtual
        virtual int saveweights(void) = 0;
        virtual int loadweights(void) = 0;
        int skipcmt(istream& s);
public:
        enum parmtype {inputs,outputs,learn,decay};
        net(){};
        net(char *s);
        net(char *s,int noparms,PARM *p);
        ~net();

        // encode and recall and "pure virtual" which makes the
        // the net class abstract
        virtual int encode(vecpair& v) = 0;
        virtual vec recall(vec &v) = 0;     // recall an output pattern given an input
        virtual float cycle(istream& s);
        virtual void train();
        int getiters(void){return iters;}
        virtual float test();   // floating point value indicates percentage correct of test
        virtual void run();

};
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
// VECMAT.H
// Vector and matrix classes
// Author:  Capt James Gainey, GEO-93D       Last Modified:  10 Sep 93
// Modified from BP.CPP  Adam Blum (1990)

#include<stdlib.h>


#include<fcntl.h>
#include<stdio.h>
#include<string.h>
#include<limits.h>
#include<ctype.h>
#include<math.h>
#include<time.h>
#include<float.h>

#ifdef __TURBOC__

#include<sys\stat.h>
#include<io.h>
#include<conio.h>
#include<alloc.h>
#elif defined(__ZTC__)
#include<dos.h>

#else

#include <gl/gl.h>
#include <gl/device.h>  // for button constant ESCKEY

#endif

#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>

#define max(a,b)       (((a) > (b)) ? (a) : (b))  // C++ doesnt have min/max
#define min(a,b)       (((a) < (b)) ? (a) : (b))

#include"debug.h"

double logistic(double activation);
#ifdef __TURBOC__
```

```
int _Cdecl intcmp(const void* i1,const void *i2);
#else
int intcmp(const void* i1,const void *i2);
#endif

// will be changed to much higher than these values
const ROWS=64;  // number of rows (length of first pattern)
const COLS=64;  // number of columns (length of second pattern)
const DELAYS= 64; //number of time delays
const MAXVEC=64; // default size of vectors

class mtrx3d;

class matrix;


class vec {
      friend ostream& operator<<(ostream& s,vec& v1);
      #ifdef __TURBOC__
      friend ostream far& operator<<(ostream far& s,vec far& v1);
      #endif
      friend class matrix;
  friend class mtrx3d;
      friend class bp;
  friend class atnn;
      friend istream& operator>>(istream& s,vec& v1);
            int n;
           float *v;
      public:
            vec(int size=MAXVEC,int val=0); // constructor
            ~vec(); // destructor
            vec(vec &v1); // copy-initializer
            int length();
            float distance(vec& A);
            vec& normalize();
            vec& normalizeon();
            vec& randomize(float initrange=1.0);
            vec& scale(vec& minvec,vec& maxvec);
            float d_logistic(); // dot product of vector and complement
            float maxval();
            vec& garble(float noise);
            vec& operator=(const vec& v1); // vector assignment
            vec operator+(const vec& v1);          // vector addition
            vec operator+(const float d);
            vec& operator+=(const vec& v1); // vector additive-assignment
```

```
        // supplied for completeness, but we don't use this now
        vec& operator*=(float c); // vector multiply by constant
        // vector transpose multiply needs access to v array
        int operator==(const vec& v1);
        float operator[](int x);
        int vec::maxindex();
        vec& getstr(char *s);
        void putstr(char *s);

        vec operator-(const vec& v1);   // vector subtraction
        vec operator-(const float d);   // subtraction
        float operator*(const vec& v1); // dot-product
        vec operator*(float c); // multiply by constant
        vec& sigmoid(vec& thresh);
        vec& set(int i,float f=0);

        int load(FILE *f);
        int save(FILE *f);
}; //end vector class

class vecpair;


class matrix {
    // we only allow access here to improve backpropagation's performance
    friend ostream& operator<<(ostream& s,matrix& m1);
    friend istream& operator>>(istream& s,matrix& m1);
    protected:
        float **m; // the matrix representation
        int r,c; // number of rows and columns
    public:
        // constructors
        matrix(int n=ROWS,int p=COLS,float range=0);
        matrix(int n,int p,float value,float range);
        matrix(int n,int p,char *fn);
        matrix(const vecpair& vp);
        matrix(vec& v1,vec& v2);
        matrix(matrix& m1); // copy-initializer
        ~matrix();
        int depth();
        int width();
        matrix& operator=(const matrix& m1);
        matrix& operator=(const vecpair& v);
        matrix operator+(const matrix& m1);
```

```
        vec operator*(vec& v1);
        vec colslice(int col);
        vec rowslice(int row);
        void insertcol(vec& v,int col);
        void insertrow(vec& v,int row);
        int closestcol(vec& v);
        int closestrow(vec& v);
        int closestrow(vec& v,int *wins,float scaling);
        int load(FILE *f);
        int save(FILE *f);
        float getval(int row,int col);
        void setval(int row,int col,float val);
        void initvals(const vec& v1,const vec& v2,
              const float rate=1.0, const float momentum=0.0);

        matrix& operator+=(const matrix& m1);
        matrix& operator*(const float d);
        matrix& operator*=(const float d);

}; // end matrix class



class vecpair {
  friend class matrix;
     friend istream& operator>>(istream& s,vecpair& v1);
     friend ostream& operator<<(ostream& s,vecpair& v1);
  friend matrix::matrix(const vecpair& vp);
        int flag; // flag signalling whether encoding succeeded
     public:
        vec *a;
        vec *b;
        vecpair(int n=ROWS,int p=COLS,int val=0); // constructor
        vecpair(vec& A,vec& B);
        vecpair(const vecpair& AB); // copy initializer
        ~vecpair();
        vecpair& operator=(const vecpair& v1);
        int operator==(const vecpair& v1);
        vecpair& scale(vecpair& minvecs,vecpair& maxvecs);
};

class mtrx3d{
     friend class matrix;
  friend istream& operator>>(istream& s,mtrx3d& m3d1);
  friend ostream& operator<<(ostream& s,mtrx3d& m3d1);
```

```
protected:
        float ***m3d; // the 3D matrix representation
        int r,c,z; // number of rows and columns
    public:
        // constructors
       mtrx3d(int n=ROWS,int p=COLS,int m=DELAYS,float range=0);
       mtrx3d(int n,int p,int m,int irange);
       mtrx3d(int n,int p,int m,float value,float range);
       mtrx3d(int n,int p,int m,char *fn);
       mtrx3d(mtrx3d& m3d1); // copy-initializer
       ~mtrx3d();
        int depth();
       int width();
       int height();
       mtrx3d& operator=(const mtrx3d& m3d1);
       mtrx3d operator+(const mtrx3d& m3d1);


        int load(FILE *f);
        int save(FILE *f);
        float getval(int row,int col,int z);
        void setval(int row,int col,int z,float val);

       mtrx3d& operator+=(const mtrx3d& m3d1);
       mtrx3d& operator*(const float d);
        mtrx3d& operator*=(const float d);

}; // end 3d matrix class
```

BIBLIOGRAPHY

1. Blum, Adam. Neural Networks in C++: An Object-Oriented Framework for Building Connectionist Systems. New York: John Wiley & Sons, Inc., 1992.

2. Bosarge, W.E. "Adaptive Processes to Exploit the Nonlinear Structure of Financial Markets" in Neural Networks in Finance and Investing. Ed. Robert R. Trippi and Efraim Turban. Chicago: Probus Publishing Co, 1993.

3. Churchland, Patricia S. and Terrence J. Sejnowski. The Computational Brain. Cambridge: The MIT Press, 1992.

4. Deutsch, Sid and Alice Deutsch. Understanding the Nervous System: An engineering Perspective. New York: The IEEE Press, 1993.

5. Gleick, James. Chaos: Making a New Science. New York: Viking Penguin, 1988.

6. Jerison, Harry J., "Paleoneurology and the Evolution of Mind," in The Workings of the Brain: Development, Memory, and Perception. Ed. Rodolfo R. Llinas. New York: W. H. Freeman and Co, 1976.

7. Kabrisky, Matthew. Professor Emeritus, School of Engineering, AFIT, Wright-Patterson AFB, OH. Personal interview. 28 May 1993.

8. Le, Phung D. Model-based 3-D Recognition System Using Gabor Features and Neural Networks. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

9. Lin, Daw-Tung, Judith E. Dayhoff and Panos A. Ligomenides. "Adaptive Time-Delay Neural Network for Temporal Correlation and Prediction.," Proceedings of SPIE Conference on Intelligent Robots and Computer Vision XI, Volume 1826. 170-181., 1992.

10. Lindsey, Randall L. Function Prediction Using Recurrent Neural Networks. MS thesis, AFIT/GEO/ENG/91D-02. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB,OH, December,1991.

11. Mish, Fredrick C., editor. Webster's Ninth New Collegiate Dictionary (First digital Edition). Boston: Merriam-Webster Inc., and NeXT Computer Inc., 1988.

12. Parker, D. Learning Logic. Technical Report TR-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology, Cambridge, MA, 1985.

13. Robbins, Anthony. Unlimited Power. New York: Ballantine Books. 1986

14. Rogers, Steven K. and Matthew Kabrisky. An Introduction to Biological and Artificial Neural Networks for Pattern Recognition. Bellingham, WA: SPIE Optical Engineering Press, 1991.

15. Rogers, Steven K. Class Lecture, EENG 548, Human Factors Engineering. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 8 September, 1993.

16. Rosenblatt, F. Principles of Neurodynamics: Perceptrons and the theory of Brain Mechanisms. Washington: Spartan Books, 1959.

17. Stright, James R. A Neural Network Implementation of Chaotic Time Series Prediction. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

18. Waibel A., T. Hanazawa, G. Hinton, K. Shikano, and K. Lang. "Phoneme recognition using time-delay neural networks," in IEEE Trans. on Acoustics., Speech, Signal Processing, 37:328-339, 1989.

19. Wan, Eric A. "Temporal Backpropagation for FIR Neural Networks," in International Joint Conference on Neural Networks, Volume 1, pages 575-580, San Diego, 1990. IEEE, New York.

20. Wan, Eric A. "Time Series Prediction Using a Connectionist Network with Internal Delay Lines," in Time Series Prediction: Forecasting the Future and Understanding the Past. Ed. Andreas S. Weigend and Neil A. Gershenfeld. Reading Mass: Addison-Wesley Publishing Company, 1993.

21. Weigend, Andreas S. and Neil A. Gershenfeld. "Results of the Time Series Prediction Competition at the Santa Fe Institute," in Proceedings of the IEEE International Conference on Neural Networks, Volume 3, 1786-1793. Piscataway, NJ: IEEE Press, 1993.

22. Werbos, P. Beyond Regression: New tools for Prediction and Analysis in the Behavioral Sciences. PhD dissertation. Harvard University, 1974.

23. Williams, Ronald J. and David Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," Neural Computation, 1: 270-280 (1989).

24. Zipser, David. "A Subgrouping Strategy that Reduces Complexity and Speeds Up Learning in Recurrent Networks," Neural Computation, 1: 552-558 (1990).

VITA

Captain James C. Gainey, Jr. was born on 1 September 1963 at McClellan AFB, California . In 1981, he graduated from Plano Senior High School in Plano, Texas. He attended Texas Tech University and received his Bachelor of Science in Electrical Engineering in December 1984. He enlisted in the United States Air Force through the College Senior Engineering Program in 1984. As a Distinguished Graduate of Officer Training School, he received a regular commission and was assigned to Aeronautical System Division at Wright-Patterson Air Force Base, Ohio. He served four years as lead Optical Systems Engineer on several successfully deployed Electronic Combat and Reconnaissance systems.

In 1989, Captain Gainey attended Undergraduate Pilot Training at Williams Air Force Base, Arizona for 51 weeks of the 52 week program. His love for flying and desire to make the best use of his pilot training experience landed him at Detachment 3, 4443rd Test and Evaluation Group, Mountain Home Air Force Base, Idaho. During 18 months as chief engineer for the Air Force's only EF-111A operational flight test unit, Captain Gainey developed an electronic countermeasures effectiveness testing program. During Desert Storm, he provided reassurance to aircrew members that their systems were fully operational prior to deployment. Captain Gainey entered the School of Engineering, Air Force Institute of Technology in May 1992.

Captain Gainey married his beautiful wife, Terri Rochelle, on June 27, 1986. Since then, the Lord has blessed them with happiness and an ever-growing love for each other.

Permanent address: 8300 Rhineway

Centerville, Ohio 45458

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
PREDICTING NONLINEAR TIME SERIES

5. FUNDING NUMBERS

6. AUTHOR(S)
James C. Gainey, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GEO/ENG/93D-05

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Predicting future values of a time series has many practical uses in real-time signal processing and understanding. This thesis implements an Adaptive Time Delay Neural Network (ATNN) capable of user-defined degeneration to the more common Time Delay Neural Network (TDNN). Time delays along axons or at the synapses, which vary in biological systems, motivate this research. The ATNN/TDNN test results and time series prediction capabilities are compared to those of the Real-Time Recurrent Learning (RTRL) algorithm. To show the advantages and disadvantages of using TDNN and ATNN for prediction versus the RTRL, the networks were applied to two problems: incommensurate sum of sine waves and financial time series. These data sets represent examples of nonlinear data with known and unknown mathematical functions, respectively. Although the RTRL predicted better than the ATNN for a known predictable function, this ATNN approach proved competitive in determining the direction of the future values for this function and outperforms the RTRL on the more difficult prediction task. The ATNN program, developed in C++ with an object-oriented framework, also takes much less computation time than the RTRL during training.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Time Series Prediction, Time Delay Neural Networks, Adaptive Time Delay Neural Networks, Neural Networks	123
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL